HARVEY M. DEITEL
PAUL J. DEITEL

C++
Fondamenti di programmazione

APOGEO

VISIT…

C++
Fondamenti di programmazione

Titolo originale:
C++ How to Program, Third Edition

Autori:
Harvey M. Deitel, Paul J. Deitel

Published by arrangement with the original publisher, PRENTICE HALL, INC.,
a Pearson Education Company
Copyright © 2001 – PRENTICE HALL, INC.

Copyright per l'edizione italiana © 2001 – APOGEO srl
Viale Papiniano 38 – 20123 Milano (Italy)
Telefono: 02-481920 (5 linee r.a.) – Fax: 02-4815382
Email education@apogeonline.com
U.R.L. www.apogeonline.com

ISBN 88-7303-670-8

Traduzione di Angelo Magliocco
Impaginazione di Maurizio Picucci
Revisione di Luca Di Gaspero
Realizzazione editoriale di Spock s.a.s. di Augusto Vico e c.
Editor: Alberto Kratter Thaler
Copertina e progetto grafico: Enrico Marcandalli
Responsabile di produzione: Vitiano Zaini
Illustrazione in copertina: © SIS/Oblinski

Tutti i diritti sono riservati a norma di legge e a norma
delle convenzioni internazionali. Nessuna parte di questo libro
può essere riprodotta con sistemi elettronici, meccanici
o altri, senza l'autorizzazione scritta dell'Editore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati
dalle rispettive case produttrici.

Sommario

PREFAZIONE XI
Lo scopo di questo libro XI
Le sezioni "Pensare in termini di oggetti" XII
Il CD-ROM XIV
La metodologia di insegnamento XIV
L'apprendimento attraverso il codice XIV
L'accesso al World Wide Web XIV
Obiettivi XIV
Il codice e gli esempi XV
La programmazione orientata agli oggetti XV
Figure e immagini XV
Consigli e suggerimenti XV
Esercizi di autovalutazione XV
Esercizi XVI
Indice analitico XVI
Panoramica del libro XVI

CAPITOLO 1: INTRODUZIONE: I COMPUTER, LA PROGRAMMAZIONE E IL C++ 1
1.1 Introduzione 1
1.2 Che cos'è un computer? 3
1.3 La struttura del computer 4
1.4 L'evoluzione dei sistemi operativi 5
1.5 I personal computer, i sistemi distribuiti e i sistemi client/server 6
1.6 I linguaggi macchina, assembly e ad alto livello 6
1.7 Il C e il C++: un po' di storia 8
1.8 La libreria standard del C++ 9
1.9 Java, Internet e il World Wide Web 10
1.10 Altri linguaggi ad alto livello 10
1.11 La programmazione strutturata 11
1.12 Gli elementi fondamentali di un tipico ambiente C++ 11
1.13 Alcune considerazioni generali sul C++ e sul nostro corso 14
1.14 Un programma semplice: visualizzare una linea di testo 16
1.15 Un altro semplice programma: l'addizione di due numeri interi 20
1.16 La memoria: concetti fondamentali 24
1.17 I calcoli aritmetici 25
1.18 Prendere decisioni: gli operatori relazionali e di uguaglianza 28
1.19 Le nuove convenzioni per i file di intestazione e gli spazi dei nomi 32
1.20 Pensare in termini di oggetti: le tecniche orientate agli oggetti e UML (Unified Modeling Language™) 34
Introduzione all'analisi e alla progettazione orientate agli oggetti (OOAD) 36
Storia di UML 37
Che cos'è UML? 38

Esercizi di autovalutazione 38
Risposte agli esercizi di autovalutazione 40
Esercizi 42

CAPITOLO 2: LE STRUTTURE DI CONTROLLO 47
2.1 Introduzione 47
2.2 Gli algoritmi 47
2.3 Lo pseudocodice 48
2.4 Le strutture di controllo 48
2.5 La struttura di selezione if 51
2.6 La struttura di selezione if/else 53
2.7 La struttura iterativa while 57
2.8 Tipologie degli algoritmi di iterazione: l'iterazione controllata da un contatore 58
2.9 *Tipologie degli algoritmi di iterazione: l'iterazione controllata* da un valore sentinella 60
2.10 Tipologie di algoritmi e ridefinizione top-down: le strutture di controllo nidificate 67
2.11 Gli operatori di assegnamento 71
2.12 Gli operatori di incremento e decremento 72
2.13 Concetti fondamentali dei cicli controllati da variabili contatore 75
2.14 La struttura di iterazione for 77
2.15 Esempi di costrutti for 81
2.16 La struttura di selezione switch 85
2.17 *La struttura iterativa do/while* *91*
2.18 Le istruzioni break e continue 93
2.19 Gli operatori logici 95
2.20 Un errore tipico: confondere l'operatore di uguaglianza == con l'operatore di assegnamento = ... 98
2.21 Riepilogo dei concetti fondamentali della programmazione strutturata 100
2.22 Pensare in termini di oggetti: come individuare le classi in un problema (progetto opzionale) 106
Definizione del problema 106
Analisi e progettazione del sistema 110
I diagrammi dei casi d'uso 111
L'identificazione delle classi di un sistema 112
I diagrammi delle classi 113
I diagrammi degli oggetti *116*
Esercizi di autovalutazione 117
Risposte agli esercizi di autovalutazione 119
Esercizi 122

CAPITOLO 3: LE FUNZIONI 133
3.1 Introduzione 133
3.2 I componenti di un programma in C++ 133
3.3 Le funzioni matematiche della libreria standard 135
3.4 Le funzioni 136

3.5 La definizione di una funzione 137
3.6 *I prototipi di funzione* 141
3.7 I file di intestazione 144
3.8 La generazione di numeri casuali 146
3.9 I giochi d'azzardo e la *parola riservata* enum 151
3.10 Le informazioni di memorizzazione 155
3.11 Le regole di visibilità 157
3.12 Il concetto di ricorsione *161*
3.13 Un altro esempio di ricorsione: la serie di Fibonacci 164
3.14 Ricorsione o iterazione? 168
3.15 Le funzioni che hanno una lista di parametri vuota 170
3.16 Le funzioni in linea 171
3.17 I riferimenti e il passaggio di parametri per riferimento 173
3.18 Gli argomenti di default 177
3.19 *L'operatore unario* di risoluzione dello scope 179
3.20 L'overloading delle funzioni 180
3.21 Le funzioni generiche 182
3.22 Pensare in termini di oggetti: *come identificare gli attributi* di una classe (progetto opzionale) 184
I diagrammi di stato 186
I diagrammi delle attività *188*
Conclusione 190
Esercizi di autovalutazione 190
Risposte agli esercizi di autovalutazione 193
Esercizi 195

CAPITOLO 4: GLI ARRAY 205
4.1 *Introduzione* 205
4.2 Gli array 205
4.3 Come si dichiara un array 208
4.4 Alcuni esempi *di array* 208
4.5 Il passaggio di un array a una funzione 222
4.6 Gli algoritmi di ordinamento e gli array 227
4.7 Il calcolo di media, mediana e moda con gli array 229
4.8 Le ricerche in array: ricerca lineare e binaria 233
4.9 Gli array multidimensionali 238
4.10 Pensare in termini di oggetti: come identificare le operazioni di una classe (progetto opzionale) 244
I diagrammi di sequenza 249
Esercizi di autovalutazione 252
Risposte agli esercizi di autovalutazione 253
Esercizi 254
Esercizi sulla ricorsione 264

CAPITOLO 5: PUNTATORI E STRINGHE 265
5.1 Introduzione 265
5.2 Come si dichiarano e si inizializzano i *puntatori* *265*

5.3 Gli operatori di manipolazione dei puntatori ... 267
5.4 La chiamata per riferimento con argomenti di tipo puntatore ... 269
5.5 Privilegi di accesso e passaggio dei parametri ... 274
5.6 L'algoritmo di ordinamento a bolle con i puntatori ... 279
5.7 L'aritmetica dei puntatori ... 285
5.8 La correlazione tra puntatori e array ... 287
5.9 Gli array di puntatori ... 292
5.10 Un programma per mescolare e distribuire carte da gioco ... 293
5.11 I puntatori a funzione ... 298
5.12 Introduzione alla manipolazione di caratteri e stringhe ... 302
Caratteri e stringhe: concetti fondamentali ... 303
Le funzioni di libreria per le stringhe ... 305
5.13 Pensare in termini di oggetti: le collaborazioni tra gli oggetti (progetto opzionale) ... 312
I diagrammi delle collaborazioni ... 314
Riepilogo ... 315
Risorse in rete sull'UML ... 316
Esercizi di autovalutazione ... 320
Risposte agli esercizi di autovalutazione ... 322
Esercizi ... 323
Sezione speciale: costruite il vostro computer ... 327
Ulteriori esercizi sui puntatori ... 332
Esercizi sulla manipolazione di stringhe ... 336
Sezione speciale: esercizi avanzati sulla manipolazione di stringhe ... 338
Un progetto di manipolazione di stringhe complesso ... 341

CAPITOLO 6: LE CLASSI E L'ASTRAZIONE DEI DATI ... 343
6.1 Introduzione ... 343
6.2 Come si definisce una struttura ... 344
6.3 Come si accede ai membri di una struttura ... 345
6.4 L'implementazione del tipo di dato Time come struttura ... 346
6.5 L'implementazione del tipo di dato Time come classe ... 348
6.6 La visibilità a livello di classe e l'accesso ai membri di una classe ... 356
6.7 La separazione di interfaccia e implementazione ... 357
6.8 Il controllo dell'accesso ai membri di una classe ... 361
6.9 Le funzioni di accesso e di utilità ... 364
6.10 L'inizializzazione degli oggetti di una classe: i costruttori ... 367
6.11 I costruttori e gli argomenti di default ... 368
6.12 I distruttori ... 371
6.13 Quando sono chiamati i costruttori e i distruttori? ... 372
6.14 L'utilizzo dei dati e delle funzioni membro ... 375
6.15 Un sottile errore logico: restituire un riferimento a un dato membro private ... 380
6.16 L'assegnamento tra oggetti: la copia di default membro a membro ... 382
6.17 Ancora sul concetto di software riutilizzabile ... 384
6.18 Pensare in termini di oggetti: programmazione delle classi del simulatore (progetto opzionale) ... 384

Implementazione: visibilità ... 385
Implementazione: handle ... 386
Implementazione: i file di intestazione delle classi ... 386
Conclusione ... 395
Esercizi di autovalutazione ... 395
Risposte agli esercizi di autovalutazione ... 396
Esercizi ... 396

CAPITOLO 7: LE CLASSI: SECONDA PARTE ... 399
7.1 Introduzione ... 399
7.2 Gli oggetti e le funzioni membro costanti ... 399
7.3 Il concetto di composizione: oggetti che diventano membri di altre classi ... 408
7.4 Le funzioni e le classi friend ... 413
7.5 Il puntatore this ... 416
7.6 L'allocazione dinamica della memoria: gli operatori new e delete ... 421
7.7 I membri static di una classe ... 423
7.8 Due concetti importanti: astrazione dei dati e occultamento delle informazioni ... 429
Il tipo di dato astratto "array" ... 430
Il tipo di dato astratto "stringa" ... 431
Il tipo di dato astratto "coda" ... 431
7.9 Le classi container e gli iteratori ... 432
7.10 Le classi proxy ... 432
7.11 Pensare in termini di oggetti: l'implementazione delle classi del simulatore (progetto opzionale) ... 435
Una panoramica dell'implementazione ... 435
Implementazione del simulatore ... 436
Esercizi di autovalutazione ... 462
Risposte agli esercizi di autovalutazione ... 463
Esercizi ... 464

CAPITOLO 8: L'OVERLOADING DEGLI OPERATORI ... 467
8.1 Introduzione ... 467
8.2 L'overloading degli operatori: concetti fondamentali ... 468
8.3 Restrizioni ... 470
8.4 La progettazione delle funzioni operatore: funzioni membro o funzioni friend? ... 471
8.5 L'overloading degli operatori di inserimento/estrazione per l'I/O su stream ... 473
8.6 L'overloading degli operatori unari ... 476
8.7 L'overloading degli operatori binari ... 476
8.8 Progettazione della classe Array ... 477
8.9 Conversioni tra tipi diversi ... 489
8.10 Progettazione della classe String ... 490
8.11 L'overloading degli operatori ++ e -- ... 502

8.12 Progettazione della classe Date ... 503
Esercizi di autovalutazione ... 508
Risposte agli esercizi di autovalutazione ... 508
Esercizi ... 509

CAPITOLO 9: L'EREDITARIETA ... 517
9.1 Introduzione ... 517
9.2 Le classi base e le classi derivate ... 519
9.3 I membri protected ... 521
9.4 Il cast dei puntatori a una classe base in puntatori a una classe derivata ... 521
9.5 Utilizzo delle funzioni membro ... 527
9.6 L'overriding di membri della classe base in una classe derivata ... 528
9.7 Ereditarietà di tipo public, protected e private ... 532
9.8 Classi base dirette e indirette ... 533
9.9 *Utilizzo dei costruttori e dei distruttori nelle classi derivate* ... 534
9.10 Conversione implicita di un oggetto di una classe derivata in oggetto della classe base ... 537
9.11 Il ruolo dell'ereditarietà nell'ingegneria del software ... 539
9.12 Composizione ed ereditarietà ... 540
9.13 Le relazioni "uses a" e "knows a" ... 540
9.14 Progettazione delle classi Point, Circle e Cylinder ... 541
9.15 L'ereditarietà multipla ... 548
9.16 Pensare in termini di oggetti: come sfruttare l'ereditarietà nel simulatore di ascensore (progetto opzionale) ... 553
Esercizi di autovalutazione ... 559
Risposte agli esercizi di autovalutazione ... 560
Esercizi ... 560

CAPITOLO 10: LE FUNZIONI VIRTUALI E IL POLIMORFISMO ... 563
10.1 Introduzione ... 563
10.2 I campi di tipo e le istruzioni switch ... 563
10.3 Le *funzioni virtuali* ... 564
10.4 Le classi base astratte e le classi concrete ... 565
10.5 Il polimorfismo ... 566
10.6 Progettazione di un libro paga elettronico ... 568
10.7 L'aggiunta di nuove classi e il binding dinamico ... 579
10.8 I distruttori virtuali ... 579
10.9 L'ereditarietà di interfaccia e di implementazione ... 580
10.10 L'implementazione di polimorfismo, funzioni virtuali e binding dinamico ... 588
Esercizi di autovalutazione ... 591
Risposte agli esercizi di autovalutazione ... 592
Esercizi ... 592

CAPITOLO 11: GLI STREAM DI INPUT/OUTPUT DEL C++ ... 595
11.1 Introduzione ... 595
11.2 Gli stream ... 596

I file di intestazione della libreria iostream ... 597
Le classi e gli oggetti che effettuano l'input/output su stream ... 597
11.3 L'output su stream ... 599
L'operatore di inserimento nello stream ... 599
Utilizzo degli operatori di inserimento/estrazione in cascata ... 601
L'output delle variabili di tipo char* ... 602
L'output di caratteri tramite put ... 602
11.4 L'input da stream ... 603
L'operatore di estrazione dallo stream ... 603
Le funzioni membro get e getline ... 606
Altre funzioni membro di istream: peek, putback e ignore ... 608
L'I/O type-safe ... 608
11.5 L'input/output non formattato delle funzioni read, gcount e write ... 609
11.6 I manipolatori di stream ... 609
La base dei numeri interi su uno stream: dec, oct, hex e setbase ... *610*
La precisione dei valori a virgola mobile: precision e setprecision ... 610
L'ampiezza dei campi: setw e width ... 612
I manipolatori definiti dall'utente ... 614
11.7 I valori di stato della formattazione ... 615
I flag di stato della formattazione ... 615
Gli zero in coda e i punti decimali: ios::showpoint ... 616
La giustificazione: ios::left, ios::right e ios::internal ... 617
Il riempimento dei campi: fill e setfill ... 619
La base dei numeri interi: ios::dec, ios::oct, ios::hex e ios::showbase ... 620
I numeri a virgola mobile e la notazione scientifica: ios::scientific e ios::fixed ... 621
Il controllo delle lettere maiuscole/minuscole: ios::uppercase ... 622
L'attivazione e la disattivazione dei flag di formattazione: flags, setiosflags e resetiosflags ... 622
11.8 I valori di stato degli errori in uno stream ... 624
11.9 Il collegamento di uno stream di output a uno stream di input ... 626
Esercizi di autovalutazione ... 626
Risposte agli esercizi di autovalutazione ... 629
Esercizi ... 631

APPENDICE A: RIEPILOGO DEGLI OPERATORI ... 633

APPENDICE B: L'INSIEME DEI CARATTERI ASCII ... 635

APPENDICE C: I SISTEMI DI NUMERAZIONE ... 637
C.1 Introduzione ... 637
C.2 Usare i numeri ottali ed esadecimali per abbreviare i numeri binari ... 640
C.3 La conversione dei numeri ottali ed esadecimali in binari ... 642
C.4 La conversione da binario, ottale o esadecimale in decimale ... 642
C.5 La conversione da decimale a binario, ottale o esadecimale ... 643
C.6 I numeri binari negativi: la notazione in complemento a due ... 644

Esercizi di autovalutazione 646
Risposte agli esercizi di autovalutazione 647
Esercizi 648

APPENDICE D: IL C++ SU INTERNET 649
D.1 Risorse 649
D.2 Tutorial 649
D.3 FAQ (risposte alle domande più comuni) 650
D.4 comp.lang.c++ 650
D.5 Compilatori 651
D.6 Standard Template Library 652
Tutorial 652
Riferimenti 652
FAQ 652
Articoli, libri e interviste 653
Software 653

INDICE ANALITICO 655

Prefazione

Benvenuti nel mondo del C++! Questo libro è stato scritto da un padre, Harvey M. Deitel, e da suo figlio, Paul J. Deitel. Il padre ha programmato e/o insegnato la programmazione per 38 anni, mentre il figlio ha programmato e/o insegnato la programmazione per 18. Il padre programma e insegna grazie all'esperienza maturata; il figlio, invece, programma e insegna grazie a un'inesauribile riserva di energia. Il padre punta alla chiarezza, il figlio alle prestazioni. Il padre cerca l'eleganza e la bellezza, il figlio vuole soprattutto risultati concreti. Insieme hanno cercato di realizzare un libro che speriamo possiate trovare utile, completo e divertente.

Lo scopo di questo libro

Il Professor Harvey M. Deitel ha tenuto corsi universitari introduttivi sulla programmazione per 20 anni, puntando soprattutto a insegnare come sia possibile sviluppare programmi ben scritti e ben strutturati. Molti dei suoi insegnamenti riguardano i concetti fondamentali della programmazione, con particolare enfasi sull'utilizzo efficace delle strutture di controllo e sulla funzionalità. Questi argomenti vengono presentati all'interno di questo libro esattamente nel modo in cui il Professor Deitel li ha sempre proposti ai propri studenti universitari.

In base alla nostra esperienza, gli studenti affrontano i temi presentati in questi capitoli nello stesso modo in cui affrontano i corsi di Pascal. C'è, comunque, un'importante differenza: gli studenti sono estremamente motivati dal fatto che stanno imparando un linguaggio che potranno utilizzare immediatamente quando lasceranno l'università e affronteranno il mondo del lavoro. Tutto questo aumenta notevolmente il loro entusiasmo nei confronti del C++, nonostante ci sia moltissimo da imparare.

Il nostro obiettivo era chiaro: offrire un testo sul C++ destinato ai corsi universitari a livello introduttivo, per quegli studenti che non hanno alcuna esperienza nell'ambito della programmazione, pur fornendo, allo stesso tempo, la completezza teorica e pratica richiesta ai tradizionali corsi avanzati di C++. Con questo proposito abbiamo scritto i due volumi *C++ Fondamenti di programmazione* e *C++ Tecniche avanzate di programmazione;* nell'insieme, un'opera assai completa su questo linguaggio.

Un importante elemento caratterizzante il testo è UML (Unified Modeling Language), uno strumento di rappresentazione grafica dei sistemi che utilizziamo nell'analisi completa, dalla definizione all'implementazione, di un caso complesso di progettazione orientata agli oggetti. La nostra sensazione è che i testi introduttivi di programmazione trascurino i progetti orientati ad oggetti di portata più vasta, quindi vogliamo raccomandare lo studio di questo caso opzionale, perché migliorerà sensibilmente l'approccio alla progettazione degli studenti del primo anno. Essi, infatti, avranno l'opportunità di navigare da subito in più di 1000 linee di codice C++ scritte e analizzate attentamente da revisori accademici e professionisti del settore.

Alla fine dei primi sette capitoli e del Capitolo 9 troverete una sezione intitolata "Pensare in termini di oggetti", il cui scopo è presentare gradualmente la progettazione orientata agli oggetti per mezzo di UML. UML è oggi lo schema più utilizzato per rappresentare graficamente i sistemi orientati agli oggetti. UML è un linguaggio grafico complesso e prevede numerose funzionalità, di cui noi vogliamo presentare soltanto un sottoinsieme. Le funzionalità che analizziamo servono per guidare il lettore nella sua prima esperienza di progettazione, rivolta per l'appunto a programmatori principianti. Lo studio di questo progetto ci accompagna per diversi capitoli fino alla sua implementazione, per cui esitiamo a chiamarlo soltanto un esercizio, perché è piuttosto un'esperienza di programmazione che si conclude solo con l'analisi del codice C++ completo.

Nei primi cinque capitoli del testo ci concentriamo sulla metodologia "convenzionale" che prende il nome di programmazione strutturata, perché gli oggetti che creeremo in seguito saranno composti in parte da porzioni di codice strutturato. Ognuno di questi capitoli termina con una sezione intitolata "Pensare in termini di oggetti", in cui presentiamo un'introduzione alla programmazione orientata agli oggetti utilizzando UML (Unified Modeling Language). Queste sezioni hanno lo scopo di aiutare gli studenti a sviluppare una mentalità orientata agli oggetti, in modo tale che sappiano impiegare da subito i concetti di programmazione orientata agli oggetti illustrati a partire dal Capitolo 6. Nella sezione "Pensare in termini di oggetti" del Capitolo 1 introduciamo i concetti fondamentali e la terminologia, mentre nelle sezioni analoghe dei Capitoli dal 2 al 5 affrontiamo questioni più sostanziali, affrontando un problema complesso con le tecniche di progettazione orientata agli oggetti (OOD, dall'inglese Object Oriented Design). In queste sezioni analizziamo la tipica definizione di un problema, che specifica il sistema desiderato, e determiniamo gli oggetti da implementare, gli attributi ad essi necessari, i comportamenti che dovranno esibire e il tipo di interazione che dovranno avere tra loro perché il sistema funzioni come specificato. Questi discorsi vengono prima di qualsiasi cenno alla programmazione orientata agli oggetti in C++. Nelle sezioni "Pensare in termini di oggetti" dei Capitoli 6, 7 e 9 discutiamo l'implementazione in C++ del sistema orientato agli oggetti che avremo progettato nei capitoli precedenti.

Questo è il progetto più complesso ed esteso del libro, e noi pensiamo che per gli studenti possa costituire un'esperienza significativa di progettazione e implementazione. L'estensione del progetto ci ha costretto a includere argomenti che non riprendiamo in nessun'altra sezione del libro, tra cui l'interazione tra oggetti, gli handle, pregi e difetti di riferimenti e di puntatori e infine le dichiarazioni anticipate, che consentono di evitare il problema dell'inclusione circolare dei file. Questo progetto rivelerà la sua utilità agli studenti quando dovranno affrontare i reali problemi dell'industria.

Il testo segue lo standard ANSI del C++; tenete presente che molte funzionalità previste nell'ANSI non sono implementate nelle versioni precedenti C++. Per avere informazioni più dettagliate sul linguaggio vi conviene consultare il manuale di riferimento del vostro sistema o procurarvi una copia del documento ANSI/ISO 9899: 1990, "American National Standard for Information Systems-Programming Language C", dell'American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

Le sezioni "Pensare in termini di oggetti"

Nel Capitolo 2 iniziamo la prima fase di progettazione orientata agli oggetti (OOD) del simulatore di ascensore identificando le classi che lo implementeranno. Introduciamo inoltre UML, con i casi d'uso, i diagrammi delle classi e degli oggetti e i concetti di associazione, molteplicità, composizione, ruolo e collegamento.

Nel Capitolo 3 determiniamo la maggior parte degli attributi necessari alle classi che abbiamo individuato. Continuiamo inoltre a introdurre UML, con i diagrammi di stato e delle attività, insieme con i concetti di evento e azione e la loro correlazione ai suddetti diagrammi.

Nel Capitolo 4 determiniamo la maggior parte delle operazioni (comportamenti) delle classi individuate. Introduciamo inoltre i diagrammi di sequenza di UML e il concetto di "messaggio inviato a un oggetto"

Nel Capitolo 5 determiniamo la maggior parte delle collaborazioni (interazioni tra oggetti) che servono a implementare il sistema, rappresentandole anche con i diagrammi delle collaborazioni UML. A fine sezione abbiamo inserito una bibliografia e un elenco di risorse in rete su UML, che mettono a disposizione tra le altre cose le specifiche di UML 1.3 e materiali come tutorial, FAQ, articoli e software.

Nel Capitolo 6 utilizziamo il diagramma delle classi UML sviluppato nelle sezioni precedenti per determinare i file di intestazione C++ che definiranno le classi. Introduciamo inoltre il concetto di *handle a un oggetto* e ne iniziamo a studiare l'implementazione in C++.

Nel Capitolo 7 presentiamo il codice C++ completo del programma di simulazione, circa 1000 linee di codice, e lo analizziamo in dettaglio. Il codice deriva direttamente dalla progettazione UML delle sezioni precedenti e tiene conto di tutte le "buone abitudini di programmazione"che predichiamo, tra cui l'utilizzo di dati membro e funzioni `static` e `const`. Parliamo inoltre dell'allocazione dinamica della memoria, della composizione e dell'interazione tra oggetti per mezzo degli handle, ed infine del problema dell'inclusione circolare dei file che si può evitare grazie alle dichiarazioni anticipate.

Nel Capitolo 9 introduciamo il concetto di ereditarietà nel programma presentato nel Capitolo 7. Suggeriamo anche ulteriori miglioramenti al codice, ma li lasciamo agli studenti come esercizio, perché possano cimentarsi con gli strumenti di progettazione con cui avranno familiarizzato nelle sezioni precedenti.

Speriamo sinceramente che questo progetto possa esseree un'esperienza di studio stimolante e utile per studenti e docenti. Da parte nostra abbiamo cercato di rendere l'intero procedimento di progettazione il più graduale possibile. Inoltre il programma che presentiamo utilizza diverse nozioni chiave della programmazione, tra cui le classi, gli oggetti, l'incapsulamento, la visibilità, la composizione e l'ereditarietà. Vi saremo grati per tutti i commenti, le critiche e i suggerimenti che vorrete inviarci all'indirizzo `deitel@deitel.com`.

Il CD-ROM

- Tutti i *programmi di esempio* sono contenuti nel CD-ROM allegato al volume. Ciò consente ai docenti di preparare le lezioni e agli studenti di padroneggiare il linguaggio più rapidamente. Gli esempi si possono anche scaricare dal sito `www.deitel.com` o dal booksite abbinato a questo libro (`www.apogeonline.com/education/booksite`). Per estrarre i sorgenti dai file .ZIP occorre utilizzare un programma di decompressione come WinZip (`http:// www.winzip.com/`) o PKZIP (`http:// www.pkware.com/`), che sono in grado di ricreare la struttura originaria delle directory. L'estrazione dei file dovrebbe avvenire su una directory distinta (ad es. *cpphtp3e_examples*).
- Il CD-ROM contiene *Microsoft Visual C++ 6 Introductory Edition software*: si tratta di un software che consente agli studenti di modificare, compilare ed effettuare il debugging dei programmi in C++. Inoltre il sito web `www.deitel.com` contiene un breve tutorial su *Visual C++ 6* in formato .PDF consultabile gratuitamente.

La metodologia di insegnamento

Questo libro contiene un'ampia varietà di esempi, esercizi e progetti che prendono spunto da molte situazioni e che offrono agli studenti la possibilità di risolvere problemi reali. Il libro prende in esame i principi della progettazione del software, insistendo sull'importanza della chiarezza dei programmi ed evitando l'uso di terminologia complessa a favore di esempi chiari e diretti, il cui codice sia stato collaudato sulle piattaforme C++ più diffuse.

L'apprendimento attraverso il codice

Il libro presenta una grande quantità di esempi basati sul codice. Ogni argomento viene presentato nell'ambito di un programma C++ completo e funzionante, seguito sempre da una o più finestre che mostrano l'output del programma in questione. Viene quindi usato il linguaggio per insegnare il linguaggio, e la lettura di questi programmi offre un'esperienza molto simile alla loro esecuzione reale su di un computer.

L'accesso al World Wide Web

Tutto il codice presente nel libro si trova anche in Internet, nel booksite abbinato a questo libro, all'indirizzo `http://www.apogeonline.com/education/booksite`. Il nostro consiglio è quello di scaricare tutto il codice, eseguendo poi ogni singolo programma via via che appare nel testo. Potete anche modificare il codice degli esempi e vedere cosa succede, imparando così a programmare programmando. Tutto questo materiale è protetto da diritti d'autore, quindi utilizzatelo liberamente per studiare il C++, ma non pubblicatene alcuna parte senza l'esplicito permesso da parte degli autori e della casa editrice.

Obiettivi

Ogni capitolo inizia con la presentazione degli *Obiettivi*. Gli studenti possono così sapere in anticipo ciò che andranno ad apprendere e, alla fine della lettura del capitolo, potranno verificare se hanno raggiunto o meno questi obiettivi.

Il codice e gli esempi

Le funzionalità del C++ vengono presentate nell'ambito di programmi completi e funzionanti. Ogni programma è seguito dalle immagini degli output che vengono prodotti, così che gli studenti possano assicurarsi della correttezza dei risultati. I programmi presentati vanno da poche linee di codice a esempi composti da varie centinaia di righe. Gli studenti dovrebbero scaricare tutto il codice dal sito Web `http:\\www.apogeonline.com/education/booksite`, eseguendo poi ogni programma via via che questo viene presentato all'interno del testo.

La programmazione orientata agli oggetti

Sin dal primo capitolo cerchiamo di comprendere insieme in che cosa consiste la programmazione orientata agli oggetti, anche se l'analizziamo in maniera formale soltanto a partire dal Capitolo 6. Nei primi cinque capitoli, quindi, introduciamo la programmazione procedurale, ossia analizziamo la componente C del C++, ma iniziamo ad abituare il lettore a "pensare in termini di oggetti" con lo sviluppo progressivo di un caso concreto, presentato nei paragrafi conclusivi di ciascun capitolo. A partire dal capitolo 6 vediamo quali sono le migliorie apportate dal C++ al C e introduciamo la programmazione ad oggetti dal punto di vista formale.

Figure e immagini

Il libro offre un'ampia varietà di grafici, immagini e output di programmi. Nelle sezioni dedicate alle strutture di controllo, per esempio, appaiono vari diagrammi di flusso molto utili. [*Nota*: I diagrammi di flusso non vengono presentati come uno strumento di sviluppo, ma ricorriamo a una breve presentazione basata su questi diagrammi per specificare le singole operazioni di ogni struttura di controllo C++.]

Consigli e suggerimenti

Il libro offre molti suggerimenti riguardanti la programmazione, per aiutare gli studenti a concentrarsi sugli aspetti più importanti dello sviluppo di programmi. Questi consigli vengono forniti attraverso le sezioni chiamate *Buona abitudine*, *Errore tipico*, *Collaudo e messa a punto*, *Obiettivo efficienza*, *Obiettivo portabilità*, *Ingegneria del software*.

Buona abitudine

Il concetto più importante per chi inizia a programmare è la chiarezza, e all'interno delle sezioni *Buona abitudine* vengono presentate delle tecniche per la scrittura di programmi chiari, comprensibili e più facilmente gestibili.

Errore tipico

Tutti gli studenti che affrontano per la prima volta un nuovo linguaggio tendono a commettere frequentemente gli stessi errori. Le sezioni *Errore tipico* aiutano gli studenti a evitare di commettere i più comuni.

Collaudo e messa a punto

Queste sezioni forniscono consigli circa le attività di test e debugging dei programmi C++, anche a livello preventivo.

Obiettivo efficienza

In base alla nostra esperienza, insegnare agli studenti come scrivere programmi chiari e comprensibili deve costituire l'obiettivo più importante di qualsiasi corso di programmazione. Gli studenti, però, vogliono normalmente imparare a scrivere programmi che vengano eseguiti in modo veloce, che utilizzino poca memoria, che richiedano una minima quantità di comandi e che offrano prestazioni eccellenti. Le sezioni *Obiettivo efficienza* offrono suggerimenti su come migliorare le prestazioni dei propri programmi.

Obiettivo portabilità

Alcuni programmatori pensano che, implementando un'applicazione C++, questa sia immediatamente portabile su tutte le piattaforme; sfortunatamente, non è sempre così. Le sezioni *Obiettivo portabilità* aiutano gli studenti a scrivere codice realmente portabile, fornendo inoltre informazioni su come il C++ sia in grado di raggiungere questo elevato livello di portabilità.

Ingegneria del software

Il C++ è un linguaggio molto efficace nell'ambito della progettazione del software, e le sezioni Ingegneria del software prendono in esame gli aspetti architetturali e progettuali che influiscono sulla realizzazione di sistemi software, specialmente nel caso di sistemi su vasta scala. Molte di queste informazioni saranno utili agli studenti nei corsi più avanzati, oltre che nel mondo del lavoro.

Esercizi di autovalutazione

Il libro propone molti esercizi corredati di risposte, così che gli studenti possano prepararsi alle esercitazioni vere e proprie. Gli studenti dovrebbero essere incoraggiati a svolgere tutti questi esercizi di autovalutazione.

Esercizi

Ogni capitolo si conclude con un insieme di esercizi di vario tipo, che permettono agli insegnanti di adattare le proprie lezioni alle esigenze particolari di ogni classe. Gli esercizi verificano l'apprendimento dei concetti e dei termini più importanti, chiedono agli studenti di scrivere singole istruzioni, piccole porzioni di funzioni, funzioni e programmi completi fino ad arrivare alla costruzione di interi progetti.

Indice analitico

Alla fine del libro è possibile trovare un indice analitico completo, molto utile sia a chi legge questo libro per la prima volta che ai programmatori che lo usano come riferimento.

Panoramica del libro

Capitolo 1 – Introduzione: i computer, la programmazione e il C++. Spiega cosa sono i computer, come funzionano e in che modo possono essere programmati. Questo capitolo introduce la programmazione strutturata e spiega perché questo insieme di tecniche abbia favorito una rivoluzione nel modo di scrivere i programmi. Il capitolo traccia brevemente la storia dello sviluppo dei linguaggi di programmazione, dai linguaggi macchina, ai linguaggi assembly e ad alto livello. Si parla anche dell'origine del C++. Il capitolo include un'introduzione agli ambienti di sviluppo tipici del C++, allo sviluppo dei programmi in C++, ai processi decisionali e alle operazioni matematiche. Dopo aver studiato questo capitolo lo studente avrà un'idea su come scrivere programmi in C++ semplici ma funzionanti. In questo capitolo discutiamo anche l'enorme diffusione di Internet e del World Wide Web. Vediamo poi cosa sono gli *spazi dei nomi* e l'istruzione using, a beneficio dei lettori che lavorano su compilatori che aderiscono allo standard. Continueremo ad utilizzare i file di intestazione "vecchio stile" nei primi capitoli di questa edizione, mentre utilizzeremo il nuovo stile nel volume Tecniche avanzate, dove analizzeremo diverse novità del linguaggio. Tuttavia ci vorrà ancora qualche anno perché i vecchi compilatori vadano in cantina.. L'approccio alla tecnologia orientata agli oggetti è del tipo full immersion, grazie alle sezioni "Pensare in termini di oggetti" che ne introducono i concetti fondamentali.

Capitolo 2 – Le strutture di controllo. Questo capitolo introduce la nozione di algoritmo come procedura per risolvere un problema e spiega l'importanza di utilizzare in modo efficace le strutture di controllo per scrivere programmi comprensibili, facili da testare e da mantenere, e funzionanti anche già dalla prima compilazione. Il capitolo introduce la struttura sequenziale, le strutture di selezione (`if`, `if/else` ed `else`) e le strutture di iterazione (`while`, `do/while` e `for`). Si esamina la struttura di iterazione in dettaglio e si mettono a confronto i cicli controllati da un valore sentinella e quelli controllati da un contatore. Il capitolo illustra poi la tecnica di raffinamento passo passo, che è di importanza critica per la stesura di programmi strutturati, e presenta lo pseudocodice, uno strumento largamente utilizzato per semplificare la progettazione di un programma. I metodi e gli approcci che utilizziamo nel Capitolo 2 si possono applicare alle strutture di controllo di qualsiasi altro linguaggio, non solo del C++. Questo capitolo aiuta lo studente a sviluppare delle buone abitudini di programmazione, preparandolo ad affrontare le complesse sfide che presentano gli aspetti più avanzati del linguaggio. Il capitolo si conclude con la discussione degli operatori logici `&&` (and), `||` (or) e `!` (not). Introduciamo poi il nuovo operatore del linguaggio `static_cast`, più sicuro del cast "vecchio stile" ereditato dal C. Vogliamo segnalare il nuovo stile che abbiamo introdotto per i sorgenti in C++, più semplice e comprensibile. Nel capitolo discutiamo infine le nuove regole di visibilità per i contatori nei cicli for. Nella sezione "Pensare in termini di oggetti" iniziamo la prima fase di progettazione orientata agli oggetti (OOD) del simulatore di ascensore identificando le classi che lo implementeranno. Introduciamo inoltre UML, con i casi d'uso, i diagrammi delle classi e degli oggetti e i concetti di associazione, molteplicità, composizione, ruolo e collegamento.

Capitolo 3 – Le funzioni. Il capitolo discute la progettazione e la scrittura dei moduli di un programma. Vi illustriamo le funzioni della libreria standard, le funzioni definite dal programmatore, la ricorsione, la chiamata per valore e la chiamata per riferimento. Le tecniche presentate in questo capitolo sono essenziali per creare programmi strutturati correttamente, e sono ancora più importanti nei programmi destinati a gestire problemi del mon-

do reale. Viene presentata la strategia del "divide et impera" come metodo efficace per risolvere problemi complessi, suddividendoli appunto in componenti più semplici che interagiscono tra di loro. Gli studenti apprezzano in genere l'argomento della generazione dei numeri casuali e della simulazione, così come i giochi di dadi che fanno un utilizzo elegante delle strutture di controllo. Il capitolo fornisce un'introduzione rigorosa alla ricorsione e include una tabella che ricapitola i diversi esempi di ricorsione in questo volume e in "Tecniche avanzate". Alcuni corsi rimandano la trattazione della ricorsione ai capitoli avanzati, mentre noi crediamo che sia preferibile un approccio graduale che inizi già dai primi capitoli del testo. I 60 esercizi che terminano il capitolo ne includono alcuni classici sulla ricorsione, come la Torre di Hanoi. Questo capitolo illustra anche i cosiddetti "miglioramenti" apportati dal C++ al C, tra cui le funzioni `inline`, i parametri di riferimento, gli argomenti di default, l'operatore unario di risoluzione dello scope, l'overloading delle funzioni e i template di funzione. Alla tabella dei file di intestazione aggiungiamo anche altri file che saranno largamente utilizzati in entrambi i volumi del corso. Vi consigliamo di lavorare all'esercizio 3.54, che applica al programma dei dadi una gestione delle scommesse. Nella sezione "Pensare in termini di oggetti"determiniamo la maggior parte degli attributi necessari alle classi che abbiamo individuato. Continuiamo inoltre a introdurre UML, con i diagrammi di stato e delle attività, insieme con i concetti di evento e azione e la loro correlazione ai suddetti diagrammi.

Capitolo 4 – Gli array. Questo capitolo discute la strutturazione dei dati in array, ovvero in *gruppi di dati dello stesso tipo correlati, e presenta numerosi esempi di array ad una e più* dimensioni. È opinione diffusa che una buona strutturazione dei dati sia almeno tanto importante quanto l'utilizzo efficace delle strutture di controllo per scrivere programmi ben strutturati. Gli esempi del capitolo illustrano varie manipolazioni di array di uso comune, la creazione di istogrammi, l'ordinamento di dati, il passaggio di array a funzioni e includono un'introduzione al campo dei sondaggi e delle inchieste di mercato (con semplici cenni di statistica). L'ordinamento e la ricerca dei dati sono argomenti fondamentali di questo capitolo: si presenta l'algoritmo di ricerca binaria, che presenta un notevole miglioramento di prestazioni rispetto a quella lineare. I 38 esercizi a fine capitolo includono diversi problemi interessanti e stimolanti, come le tecniche di ordinamento avanzate, la progettazione di un sistema di prenotazione aerea automatizzata, un'introduzione alla "turtle graphics" che deve la fama al linguaggio LOGO, ed infine i problemi del "Giro del cavallo" e delle "Otto regine" che introducono il concetto di programmazione euristica, largamente impiegata nel campo dell'intelligenza artificiale. Gli esercizi si concludono con 8 problemi di ricorsione tra cui l'ordinamento per selezione, i palindromi, la ricerca lineare, la ricerca binaria, le otto regine, la visualizzazione di un array, la visualizzazione di una stringa al contrario e la ricerca del valore minimo presente in un array. In questo capitolo utilizziamo gli array in stile C che, come vedremo nel Capitolo 5, sono in realtà puntatori a locazioni di memoria consecutive. Arriveremo in seguito a vedere gli array come oggetti a tutti gli effetti: nel Capitolo 8 utilizziamo l'overloading degli operatori su una classe `Array` da cui è possibile creare oggetti array più robusti e facili da programmare rispetto a quelli "vecchio stile" del C. Nel Capitolo 9 del volume Tecniche avanzate introdurremo la classe `vector` della "Libreria standard dei template" (STL), che si utilizza in congiunzione con gli iteratori e gli algoritmi discussi in quel capitolo e che consente di vedere e trattare un array come oggetto a tutti gli effetti. Nella sezione "Pensare in termini di oggetti"determiniamo la maggior parte delle operazioni (comportamenti) delle classi individuate. Introduciamo inoltre i diagrammi di sequenza di UML e il concetto di "messaggio inviato a un oggetto"

Capitolo 5 – Puntatori e stringhe. Questo capitolo presenta una delle caratteristiche del linguaggio più potenti ma anche più difficili da padroneggiare, i puntatori. V'è una spiegazione dettagliata degli operatori di manipolazione dei puntatori, della chiamata per riferimento, delle espressioni di puntamento e dei puntatori a funzioni. C'è una stretta correlazione in C++ tra puntatori, array e stringhe, per cui introduciamo anche i concetti di base sulla manipolazione delle stringhe e illustriamo come funzionano alcune funzioni di manipolazione molto utilizzate, come `getline` (input di una linea di testo), `strcpy` e `strncpy` (copia di una stringa), strcat e strncat (concatenamento di due stringhe), strcmp e strncmp (confronto di due stringhe), `strtok` (estrazione di elementi, o token, da una stringa) e `strlen` (calcolo della lunghezza di una stringa). I 49 esercizi a fine capitolo includono la classica simulazione della gara di corsa tra la tartaruga e la lepre, gli algoritmi di mescolamento e distribuzione delle carte da gioco, l'ordinamento veloce (quicksort) ricorsivo e l'attraversamento ricorsivo di un labirinto. Abbiamo incluso anche la sezione speciale "Costruite il vostro computer", che introduce la programmazione in linguaggio macchina e guida il lettore nella progettazione e nell'implementazione di un simulatore di computer, che consente di scrivere ed eseguire dei programmi in linguaggio macchina. Questo esercizio alquanto insolito è una caratteristica del nostro corso e sarà di notevole aiuto ai lettori che vogliono comprendere come funziona realmente un computer. Troviamo che i nostri studenti generalmente si appassionano a questo progetto e spesso ne implementano miglioramenti sostanziali, alcuni dei quali sono suggeriti direttamente da noi nella sezione degli esercizi. Nel Capitolo 4 del volume Tecniche avanzate includeremo un'altra sezione speciale che guida il lettore nella progettazione di un compilatore: il linguaggio macchina prodotto da questo compilatore potrà poi essere eseguito sul simulatore creato nel Capitolo 7; i dati saranno comunicati dal compilatore al simulatore su file sequenziali, come vedremo nel Capitolo 3 del volume Tecniche avanzate. C'è una seconda sezione speciale che include esercizi di manipolazione di stringhe stimolanti come l'analisi dei testi, il word processing, la visualizzazione di dati in diversi formati, il controllo della protezione, la scrittura dell'equivalente in lettere della somma di un assegno, il codice Morse e la conversione metrica dal sistema metrico decimale a quello anglosassone. Il lettore avrà voglia di tornare su questi esercizi dopo aver studiato la classe `string` nel Capitolo 8 del volume Tecniche avanzate. Molti credono che l'argomento dei puntatori sia di gran lunga il più complesso in un corso di programmazione introduttivo. In C e nel C++ non orientato agli oggetti gli array e le stringhe sono in realtà puntatori a celle di memoria consecutive, e sono puntatori persino i nomi delle funzioni. Lo studio attento di questo capitolo dovrebbe ripagarvi di una comprensione profonda di questo complesso argomento. Ricordiamo comunque che riparleremo di array e stringhe come oggetti a tutti gli effetti più in là nel corso. Nel Capitolo 8 utilizziamo l'overloading degli operatori per creare le classi `Array` e `String`. Nel Capitolo 8 del volume Tecniche avanzate discutiamo della classe string della Libreria standard e mostriamo come manipolare gli oggetti `string`. Nel Capitolo 9 di tale volume discutiamo anche la classe `vector`. Nella sezione "Pensare in termini di oggetti" determiniamo la maggior parte delle collaborazioni (interazioni tra oggetti) che servono a implementare il sistema, rappresentandole anche con i diagrammi delle collaborazioni UML. A fine sezione abbiamo inserito una bibliografia e un elenco di risorse in rete su UML, che mettono a disposizione tra le altre cose le specifiche di UML 1.3 e materiali come tutorial, FAQ, articoli e software.

Capitolo 6 – Le classi e l'astrazione dei dati. Questo capitolo inizia lo studio della programmazione orientata agli oggetti. Questo corso è una splendida opportunità di insegnare l'astrazione dei dati nel modo che riteniamo più giusto, ovvero attraverso un linguaggio, il

C++, che è stato pensato espressamente per implementare tipi di dati astratti (ADT). Negli ultimi anni l'astrazione dei dati è diventata uno degli argomenti più importanti dei corsi introduttivi di programmazione. I capitoli 6, 7 e 8 ne includono una discussione rigorosa. Il Capitolo 6 mostra l'implementazione degli ADT come `struct`, la loro implementazione come classi in stile C++ e le ragioni per cui quest'ultima è superiore alla prima, l'accesso ai membri di una classe, la separazione di interfaccia e implementazione, l'utilizzo di funzioni di accesso e di utilità, l'inizializzazione di oggetti tramite costruttori, la loro distruzione tramite distruttori, l'assegnamento di default membro a membro e il riutilizzo del software. Gli esercizi a fine capitolo stimolano gli studenti a sviluppare classi per numeri complessi, numeri razionali, orari, date, rettangoli, interi di grandi dimensioni e gioco del tris. Generalmente gli studenti si appassionano ai programmi di gioco. Il lettore più incline alla matematica gradirà gli esercizi sulla classe `Complex` (numeri complessi), sulla classe `Rational` (numeri razionali) e sulla classe `HugeInteger` (numeri di qualsiasi dimensione). Nella sezione "Pensare in termini di oggetti"utilizziamo il diagramma delle classi UML sviluppato nelle sezioni precedenti per determinare i file di intestazione C++ che definiranno le classi. Introduciamo inoltre il concetto di handle a un oggetto e ne iniziamo a studiare l'implementazione in C++.

Capitolo 7 - Le classi: seconda parte. Questo capitolo continua lo studio delle classi e dell'astrazione dei dati. Il capitolo spiega come dichiarare e utilizzare oggetti e funzioni membro costanti, la composizione che permette di creare nuove classi che contengono come membri oggetti di altre classi, le funzioni e le classi friend che hanno diritti di accesso speciali ai dati private e protected delle classi, il puntatore this che consente ad un oggetto di conoscere il proprio indirizzo, l'allocazione dinamica della memoria, i membri static che consentono di memorizzare e manipolare dati comuni a tutti gli oggetti di una classe, e mostra infine alcuni esempi di tipi di dati astratti molto utilizzati (array, stringhe e code), di classi container e di iteratori. Gli esercizi del capitolo chiedono allo studente di sviluppare una classe "conto corrente" e una classe per insiemi di interi. Nella nostra discussione degli oggetti costanti facciamo brevemente riferimento alla nuova parola chiave mutable che, come vedremo nel Capitolo 10 del volume "Tecniche avanzate", serve per modificare l'implementazione "invisibile" degli oggetti const. Nel capitolo discutiamo anche l'allocazione dinamica della memoria con new e delete. Se new non riesce ad allocare memoria restituisce un puntatore 0 nelle vecchie versioni del C++: noi utilizziamo questo stile in tutto questo volume e nel primo capitolo del volume Tecniche avanzate. Nel Capitolo 2 di tale volume, invece, introduciamo il nuovo comportamento di new in caso di errori, cioè il "lancio di un'eccezione". Abbiamo inserito la spiegazione dei membri static in un esempio di videogioco. In tutto il libro, così come nei nostri seminari, poniamo l'accento sull'importanza di tenere nascosti i dettagli di implementazione ai client di una classe. Tuttavia i dati private sono perfettamente visibili nei file di intestazione della nostra classe, cosa che naturalmente rivela tali dettagli. Abbiamo quindi introdotto una nuova sezione sulle classi proxy, che rappresentano un meccanismo efficace per nascondere l'implementazione ai client di una classe. Nella sezione "Pensare in termini di oggetti"presentiamo il codice C++ completo del programma di simulazione, circa 1000 linee di codice, e lo analizziamo in dettaglio. Il codice deriva direttamente dalla progettazione UML delle sezioni precedenti e tiene conto di tutte le "buone abitudini di programmazione"che predichiamo, tra cui l'utilizzo di dati membro e funzioni static e const. Parliamo inoltre dell'allocazione dinamica della memoria, della composizione e dell'interazione tra oggetti per mezzo degli handle, ed infine del problema dell'inclusione circolare dei file che si può evitare grazie alle dichiarazioni anticipate.

Capitolo 8 – L'overloading degli operatori. È tra gli argomenti più importanti di un corso di C++ ed è molto apprezzato dagli studenti, perché lo trovano perfettamente conseguente alla discussione degli ADT dei Capitoli 6 e 7. L'overloading degli operatori consente di indicare al compilatore come interpretare gli operatori già esistenti quando vengono utilizzati con i nuovi oggetti creati dall'utente. Il C++ sa già come utilizzare questi operatori con gli oggetti dei tipi predefiniti come interi, numeri a virgola mobile e caratteri. Supponiamo dunque di creare una nuova classe di stringhe: che cosa potrebbe significare il segno più tra due stringhe? Molti programmatori utilizzano il segno più con le stringhe per indicare il concatenamento. Nel Capitolo 8 si spiega come "sovraccaricare" (in inglese *overload*) il segno più in modo tale che quando il compilatore lo rileva in un'espressione tra due oggetti stringa genera una chiamata alla *funzione operatore* che effettua il concatenamento. Il capitolo discute i concetti di base dell'overloading degli operatori, le restrizioni, l'overloading con le funzioni membro di una classe e le funzioni non membro, l'overloading di operatori unari e binari e la conversione tra tipi. Una caratteristica del capitolo è costituita dai diversi esempi significativi tra cui una classe di array, una di stringhe, una di date, una di interi di grandi dimensioni e una di numeri complessi (gli ultimi due compaiono negli esercizi insieme con il codice completo). Lo studente più incline alla matematica vorrà cimentarsi negli esercizi con la creazione di una classe di polinomi. Questo materiale è diverso da quello che normalmente si può trovare in corsi simili al nostro. L'overloading degli operatori è un argomento complesso ma di notevole valore aggiunto, perché consente di dare quella "rifinitura" finale alle proprie classi. Le discussioni sulla classe `Array` e sulla classe `string` sono di particolare interesse per gli studenti che hanno intenzione di studiare successivamente le classi della Libreria standard `string` e `vector`. Grazie alle tecniche descritte nei Capitoli 6, 7 e 8 è possibile creare una classe `Date` che avremmo voluto vedere utilizzata già dal 1980, perché ci avrebbe risparmiato gran parte dei problemi che abbiamo incontrato nel passaggio all'anno 2000. Gli esercizi incoraggiano lo studente a integrare l'overloading degli operatori nelle classi Complex, Rational e HugeInteger, per poter manipolare questi oggetti con dei simboli, come in matematica, piuttosto che con chiamate di funzione, come accadeva negli esercizi del Capitolo 7.

Capitolo 9 – L'ereditarietà. Questo capitolo parla di una delle funzionalità principali dei linguaggi orientati agli oggetti. L'ereditarietà è una forma di riutilizzo del software grazie alla quale è possibile sviluppare nuove classi in modo facile e rapido "assorbendo" le funzionalità di altre classi esistenti e aggiungendone di nuove. Il capitolo discute i seguenti concetti: classe base e classe derivata, membri `protected`, ereditarietà di tipo `public`, `protected` e `private`, classi base dirette e indirette, costruttori e distruttori di classe base e derivata; infine discute dell'ingegneria del software che utilizza l'ereditarietà. Il capitolo confronta l'ereditarietà (relazione "is a") con la composizione (relazione "has a") e introduce le relazioni "uses a" e "knows a". Una caratteristica di questo capitolo sta nei casi di studio significativi proposti. In particolare, un caso alquanto lungo implementa la gerarchia di classi punto/cerchio/cilindro. Il capitolo si conclude con un caso di ereditarietà multipla, una caratteristica avanzata del C++ che consente di derivare gli attributi e i comportamenti di una classe da diverse classi base. Gli esercizi chiedono allo studente di confrontare la creazione di nuove classi con l'ereditarietà e con la composizione, di estendere le diverse gerarchie di ereditarietà discusse nel capitolo, di scrivere una gerarchia per i quadrilateri, i trapezi, i parallelogrammi, i rettangoli e i quadrati ed infine di creare una gerarchia più generale di forme geometriche che distingue tra forme nel piano e forme nello spazio. Abbiamo modificato la gerarchia dell'ereditarietà relativa ai membri della

comunità universitaria per fornire un esempio di ereditarietà multipla. Nel Capitolo 10 del volume Tecniche avanzate continueremo la discussione sull'ereditarietà multipla esponendo i problemi causati dalla cosiddetta "ereditarieta romboidale" e mostrando come si possono risolvere grazie alle classi base virtuali. Nella sezione "Pensare in termini di oggetti"introduciamo il concetto di ereditarietà nel programma presentato nel Capitolo 7. Suggeriamo anche ulteriori miglioramenti al codice, ma li lasciamo agli studenti come esercizio, perché possano cimentarsi con gli strumenti di progettazione con cui avranno familiarizzato nelle sezioni precedenti.

Capitolo 10 – Le funzioni virtuali e il polimorfismo. Questo capitolo introduce un'altra funzionalità importante della programmazione orientata agli oggetti, il comportamento polimorfico. Quando diverse classi sono correlate per mezzo dell'ereditarietà a una classe base comune, ogni oggetto di una classe derivata può essere trattato come un oggetto di classe base: ciò consente di scrivere programmi generici e indipendenti dai tipi specifici delle classi derivate. Si possono gestire con uno stesso programma nuovi tipi di oggetto, e ciò contribuisce a rendere un sistema estensibile. Il polimorfismo consente di evitare la complessa logica degli `switch` nei programmi, a favore di una logica lineare più comprensibile. Lo screen manager di un videogioco, per esempio, può inviare semplicemente un messaggio "disegna" a tutti gli elementi di una lista concatenata di oggetti da disegnare sullo schermo. Ciascun oggetto saprà come disegnare se stesso. Inoltre si può aggiungere un nuovo oggetto al programma senza modificare quest'ultimo, sempre che il nuovo oggetto sappia come disegnare se stesso. Questo stile di programmazione si utilizza tipicamente per implementare le comuni interfacce grafiche per gli utenti (dette GUI). Il capitolo discute come si implementa un comportamento polimorfico con le funzioni virtuali. Si mostra la distinzione tra classi astratte, da cui non è possibile istanziare oggetti, e classi concrete, da cui ciò è possibile. Le classi astratte sono utili per fornire un'interfaccia ereditabile a classi che non sono in una gerarchia. Il capitolo presenta due casi di studio importanti sul polimorfismo: un libro paga elettronico e una seconda versione della gerarchia di forme geometriche discussa nel Capitolo 9. Gli esercizi chiedono allo studente di discutere diversi approcci e questioni concettuali, di includere classi astratte alla gerarchia di forme geometriche, di sviluppare un semplice pacchetto grafico e di modificare la classe `employee` introdotta nel capitolo. I due casi di polimorfismo mostrano una differenza di stili nell'ereditarietà: il libro paga fa un uso chiaro e "sensibile" dell'ereditarietà, mentre il secondo esempio è un esempio di "ereditarietà strutturale", che non è naturale come il primo tipo ma ugualmente corretta da un punto di vista formale e funzionale. Abbiamo deciso di includere questo secondo esempio in relazione alla sezione "L'implementazione di polimorfismo, funzioni virtuali e binding dinamico" I nostri seminari si rivolgono a programmatori esperti ed essi apprezzavano i nostri esempi di polimorfismo, ma dicevano che alla nostra presentazione mancava qualcosa: è vero, mostravamo come programmare in C++ con il polimorfismo, ma essi volevano qualcosa in più. Nello specifico ci dissero che erano preoccupati del consumo aggiuntivo di risorse che il polimorfismo inevitabilmente implicava, per cui ci chiesero di fornire loro una spiegazione più approfondita del modo in cui è implementato in C++, e di conseguenza di quanto "costa" in termini di tempo di esecuzione e di memoria. Abbiamo dato ascolto a questa loro richiesta sviluppando una sezione che illustra le *vtable* (tabelle delle funzioni virtuali) create automaticamente dal compilatore quando incontra il polimorfismo. Abbiamo applicato queste tabelle alle classi della gerarchia di forme geometriche. Abbiamo avuto conferma dal nostro pubblico che esse davano loro le informazioni necessarie per decidere caso per caso se adottare uno stile di programmazione polimorfico. Questa presentazione si trova nella Sezione 10.10 e l'illustrazione delle vtable è in Figura 10.2.

Capitolo 11 – Gli stream di input/output del C++. Questo capitolo contiene una discussione esauriente del nuovo input/output orientato agli oggetti introdotto in C++. Illustriamo le varie funzionalità di I/O del C++ tra cui l'output con l'operatore di inserimento nello stream, l'input con l'operatore di estrazione dallo stream, l'I/O orientato ai tipi di dato (un miglioramento significativo rispetto al C), l'I/O formattato e non formattato (per migliori prestazioni), i manipolatori degli stream per il controllo della base numerica di uno stream (decimale, ottale o esadecimale), l'I/O dei numeri a virgola mobile, il controllo dell'ampiezza del campo, i manipolatori definiti dall'utente, gli stati di formattazione di uno stream, gli stati di errore di uno stream, l'I/O di oggetti di tipi definiti dall'utente e il collegamento di uno stream di output a uno stream di input per assicurare che la richiesta di un input all'utente appaia effettivamente prima che il sistema si metta in attesa. L'insieme di esercizi chiede allo studente di scrivere diversi programmi che testano la maggior parte delle funzionalità di I/O discusse nel corpo del capitolo.

CAPITOLO 1

Introduzione: i computer, la programmazione e il C++

Obiettivi

- Comprendere alcuni concetti fondamentali dell'informatica
- Conoscere i diversi tipi di linguaggi di programmazione esistenti
- Conoscere un tipico ambiente di sviluppo di programmi in C++
- Imparare a scrivere semplici programmi in C++
- Imparare a utilizzare le istruzioni più semplici per l'input e l'output
- Conoscere i tipi di dato fondamentali
- Imparare a utilizzare gli operatori aritmetici
- Capire le precedenze degli operatori aritmetici
- Imparare a scrivere semplici istruzioni decisionali

1.1 Introduzione

Benvenuti nel nostro corso sul C++! Abbiamo lavorato duramente per scrivere quest'opera, e speriamo che il risultato sia il più possibile completo, divertente e interessante. Il C++ è un linguaggio difficile, ed i testi di programmazione in C++ sono normalmente rivolti a programmatori esperti. Il nostro testo si discosta da questa filosofia ed è rivolto sia a persone che hanno un background tecnico ma poca esperienza di programmazione, che a programmatori esperti che vogliono approfondire la loro conoscenza del linguaggio.

Com'è possibile parlare ad un pubblico così vasto? La risposta sta nel fatto che cerchiamo di insegnare come programmare con *chiarezza* per mezzo delle ben note tecniche di *programmazione strutturata* e di *programmazione orientata agli oggetti*. Anche chi non ha esperienza di programmazione inizia dunque nel modo giusto. Abbiamo cercato inoltre di rendere il testo chiaro, diretto e ricco di illustrazioni. Un'altra caratteristica importante è il cospicuo numero di programmi in C++ completi e funzionanti: oltre ai listati includiamo anche l'output prodotto durante l'esecuzione. Tutte le caratteristiche del linguaggio sono dunque contestualizzate in programmi reali secondo il cosiddetto approccio *live-code*.

I primi cinque capitoli del libro introdurranno i concetti fondamentali relativi ai computer, alla programmazione e al linguaggio C++. Questi capitoli costituiscono le basi indispensabili per continuare il corso. È probabile che i programmatori esperti vogliano legge-

re questi primi capitoli molto velocemente per passare subito ai successivi, che troveranno interessanti e rigorosi.

Diversi programmatori esperti ci hanno confidato di apprezzare il nostro approccio alla programmazione strutturata. Infatti, anche persone che da molto tempo programmano in linguaggi strutturati (come il C o il Pascal), ma che non hanno mai affrontato una trattazione organica della programmazione strutturata non sono in grado di scrivere il miglior codice possibile per le loro applicazioni. Rivisitando la programmazione strutturata, nei primi capitoli di questo libro, essi saranno in grado di migliorare il proprio stile di programmazione, anche in C e in Pascal. In definitiva, qualsiasi esperienza abbiate alle spalle, troverete questo libro di sicura utilità.

Molta gente ha una conoscenza di base dei computer e di quello che è possibile fare con essi. Grazie al nostro libro imparerete a programmare i computer per fare queste cose. Il controllo dei computer avviene per mezzo del *software*: un insieme di istruzioni che *ordinano al computer di effettuare delle azioni e di prendere delle decisioni*. Il computer nella sua fisicità, prende invece il nome di *hardware*.

Oggi i computer stanno invadendo tutte le aree produttive. In un'epoca in cui i prezzi salgono drasticamente, il costo dei computer sta invece scendendo di continuo, grazie al rapido sviluppo della tecnologia sia hardware che software. Computer che 25 anni fa occupavano enormi stanze e costavano miliardi di lire possono essere costruiti oggi su chip di silicio più piccoli di un'unghia che costano al massimo qualche migliaio di lire. Sorprendentemente, il silicio è uno dei materiali più comuni sulla Terra (la sabbia, ad esempio, è costituita proprio di silicio). I chip di silicio hanno reso talmente economica la tecnologia dei computer che, ad oggi, contiamo circa 200 milioni di computer in tutto il mondo, dislocati in industrie, aziende, governi e abitazioni private; presumibilmente, tale numero raddoppierà nel giro di pochi anni.

Il C++ è uno dei linguaggi di programmazione più utilizzati al giorno d'oggi. Il linguaggio e lo stile di programmazione seguiti in questo libro si conformano allo standard americano ANSI (*American National Standards Institute*) e allo standard internazionale ISO (*International Standards Organization*). *Una delle ragioni per le quali questo libro catturerà* il vostro interesse è la possibilità di imparare due linguaggi in un colpo solo, poiché il C++ è un'estensione del C.

I vostri colleghi hanno imparato probabilmente una metodologia di programmazione nota con il nome di *programmazione strutturata*. Oltre ad essa, voi imparerete anche la *programmazione orientata agli oggetti:* una metodologia più recente ed efficace, il cui uso si imporrà nel giro di pochi anni. In questo corso avrete modo di creare e interagire con molti *oggetti*. Molti di essi però sono strutturati internamente nel modo migliore possibile proprio grazie alle tecniche di programmazione strutturata. Essa esprime al meglio anche la logica che è insita nella manipolazione di tali oggetti.

Un ulteriore motivo per cui proponiamo entrambe le metodologie è il fatto che stiamo *assistendo ad una massiccia migrazione da sistemi che si basano sul C verso sistemi basati* sul C++. Il C è utilizzato da circa un quarto di secolo, ed il suo utilizzo è aumentato notevolmente negli ultimi anni: dunque ancora una gran quantità dei programmi in circolazione sono scritti in C. In seguito alla diffusione del C++, però, i produttori di software *hanno iniziato a ripensare le proprie strategie: inizialmente, essi convertono i propri siste-*

mi scritti in C sfruttando il fatto che il C++ è un'estensione del C. In seguito, cominciano a utilizzare le caratteristiche proprie del C++ (che sono in definitiva dei miglioramenti di concetti già presenti in C), ed infine iniziano a impiegare tecniche di programmazione orientata agli oggetti in C++, in modo da sfruttare appieno i benefici offerti dal linguaggio.

Un fenomeno interessante riguarda il mercato dei linguaggi: i produttori vendono più spesso prodotti C/C++ integrati, anziché i due prodotti *separatamente. Ciò avviene per* consentire agli utenti di continuare a utilizzare il C man mano che acquisiscono familiarità con le novità del C++.

Il C++ è oggi il linguaggio di programmazione più comune. Vi chiederete forse se è possibile insegnarlo in un primo corso di programmazione, che sarebbe la destinazione naturale di questo libro? Noi pensiamo di sì: cinque anni fa abbiamo affrontato una sfida analoga con il nostro testo di C contro il Pascal, quando questo era il linguaggio di elezione nei corsi di informatica di base. Il risultato è stato che centinaia di Università in tutto il mondo hanno adottato il nostro testo di programmazione in C. Inoltre, l'efficacia didattica di tali corsi è risultata paragonabile a quelli sul Pascal: non si sono osservate differenze significative, eccetto per il fatto che gli studenti sono maggiormente motivati ora, perché sanno già che utilizzeranno il C, piuttosto che il Pascal, nei loro corsi superiori e nel loro lavoro. Gli studenti che imparano il C sanno anche che avranno le basi per imparare il C++ e il nuovo linguaggio di Internet *Java*, che si fondano sui concetti della programmazione orientata agli oggetti.

Nei primi cinque capitoli verranno presentati i concetti *della programmazione strutturata* in C++, verrà presentata la porzione di linguaggio C contenuta nel C++ e i miglioramenti presenti nel C++; nella parte centrale del libro affronteremo la programmazione orientata agli oggetti in C++. Non vogliamo però che aspettiate fino al Capitolo 6 per avere una conoscenza minima degli oggetti e dei concetti *relativi. Perciò, alla fine dei primi* cinque capitoli, includeremo sempre una sezione intitolata "Pensare in termini di oggetti" in cui introdurremo i concetti e la terminologia di base della programmazione orientata agli oggetti. In questo modo, quando arriveremo al Capitolo 6, che spiega le classi e l'astrazione dei dati, sarete già pronti ad utilizzare il C++ *per creare oggetti e scrivere* programmi orientati agli oggetti.

Questo primo capitolo si divide in tre parti. La prima introduce i concetti di base relativi ai computer e alla programmazione. Nella seconda vi chiediamo di cominciare subito a scrivere alcuni semplici programmi in C++. Nell'ultima cominciamo a "Pensare in termini di oggetti"

Allora, salite pure a bordo! Siete all'inizio di un percorso che noi riteniamo interessante e gratificante. Speriamo vivamente che alla fine sarete d'accordo con noi!

1.2 Che cos'è un computer?

Un computer è un dispositivo in grado di effettuare *operazioni matematiche e di prendere* decisioni a velocità milioni di volte superiori rispetto agli esseri umani. Per esempio molti computer moderni sono in grado di effettuare cento milioni di addizioni al secondo. Chiunque voglia sfidarli munito di carta e penna potrebbe rimanere inchiodato alla scrivania per decenni prima di completare lo stesso compito

I *supercomputer* di oggi possono effettuare anche centinaia di miliardi di addizioni al secondo, quasi altrettante quante ne effettuerebbero in un anno centinaia di migliaia di persone! Non solo: nei laboratori di ricerca sono stati già sviluppati dei prototipi di computer che lavorano a un ritmo di migliaia di miliardi di operazioni al secondo!

I computer elaborano *dati* sotto il controllo di insiemi di istruzioni detti *programmi*. Questi impongono al computer di eseguire determinate operazioni, e sono scritti da persone chiamate *programmatori*. Le varie unità che costituiscono un computer vengono chiamate *hardware*: esse sono la tastiera, lo schermo, il mouse, le unità a dischi, la memoria, l'unità CD-ROM e una o più unità di elaborazione. I programmi che vengono eseguiti (in gergo, "girano", dall'inglese *to run*) su un computer costituiscono quello che viene chiamato *software*. Grazie al miglioramento dei sistemi di produzione, i costi dell'hardware negli ultimi anni sono risultati in continuo calo, al punto tale che i computer sono diventati a tutti gli effetti dei comuni beni di consumo. Al contrario, i costi relativi allo sviluppo del software sono in aumento. Ciò è dovuto alla necessità di applicazioni sempre più potenti e complesse senza che dall'altra parte vi sia stato un miglioramento sensibile delle tecnologie di sviluppo del software. Questo libro vi insegnerà delle metodologie di sviluppo del software già sperimentate, in grado di ridurre i costi di sviluppo: stiamo parlando della programmazione strutturata, della definizione dei programmi grazie a tecniche di raffinamento top-down, della decomposizione funzionale e della programmazione orientata agli oggetti.

1.3 La struttura del computer

Tralasciando le differenze d'aspetto tra i vari computer, possiamo considerarli astrattamente come composti di sei *unità logiche* di seguito descritte:

1. *Unità di input*. Riceve le informazioni (sia dati che programmi) da varie *periferiche di input* e le mette a disposizione delle altre unità, in modo da consentirne l'elaborazione. Le informazioni sono generalmente immesse nei computer tramite la tastiera e il mouse. In futuro parte dei dati potranno essere immessi parlando direttamente al computer oppure acquisendo delle immagini tramite lo scanner.
2. *Unità di output*. Invia le informazioni elaborate alle varie *periferiche di output* per renderle disponibili all'esterno. Attualmente, la maggior parte delle informazioni sono inviate allo schermo, stampate su carta o utilizzate per il controllo di altre periferiche.
3. *Memoria*. Conserva i dati e i programmi, consentendone un rapido accesso dalle componenti di elaborazione. La sua capacità è relativamente bassa, trattandosi solamente di un deposito temporaneo di dati. Contiene le informazioni immesse in precedenza nell'unità di input, e le rende disponibili immediatamente per l'elaborazione. Contiene anche le informazioni già elaborate, in modo che possano essere convogliate successivamente sulle unità di output. Viene anche detta *memoria primaria*.
4. *Unità aritmetico-logica (ALU)*. È il cuore di calcolo del computer. Effettua addizioni, sottrazioni, moltiplicazioni e divisioni. Prevede anche dei meccanismi decisionali, che consentono al computer di confrontare due dati presenti in memoria (ad esempio per determinare se essi sono uguali oppure no).
5. *Unità centrale di elaborazione (CPU)*. Coordina le operazioni e ha compiti di supervisione sulle altre unità. È la CPU a decidere quando le informazioni presenti nelle unità di input debbano essere trasferite in memoria, quando la ALU debba elaborarle, e quando i risultati dell'elaborazione debbano essere convogliati verso le unità di output.

6. *Unità di memorizzazione secondaria*. Anche questa unità memorizza i dati: a differenza della memoria primaria e un'unità di memorizzazione a lungo termine e ad alta capacità. I programmi e i dati che non vengono utilizzati dalle altre unità vengono posti normalmente su unità di memorizzazioni secondarie, come i dischi, finché non se ne ha nuovamente bisogno. L'accesso a queste unità è molto più lento di quello alla memoria primaria, mentre il costo di un'unità di memorizzazione secondaria è molto più basso di quello della memoria primaria.

1.4 L'evoluzione dei sistemi operativi

I primi computer erano in grado di svolgere soltanto un'elaborazione per volta (ad esempio calcolare le paghe dei dipendenti oppure fare le previsioni del tempo) dedicandosi interamente a questo compito. Questa forma di elaborazione è anche detta *elaborazione batch monoutente*: il computer esegue un solo programma alla volta ed elabora i dati in raggruppamenti di programmi detti *batch* (o *lotti*). In questi sistemi primitivi gli utenti consegnavano le proprie schede perforate, contenenti dati e programmi, agli operatori del centro di calcolo che decidevano come raggruppare le elaborazioni e quando mandarle in esecuzione; i tempi di elaborazione potevano variare da un minimo di alcune ore a diversi giorni.

In seguito furono inventati i cosiddetti *sistemi operativi*, allo scopo di rendere più agevole l'uso dei computer. I primi sistemi operativi gestivano la successione dei compiti immessi dagli utenti e le transizioni tra un compito e l'altro. In questo modo si recuperava il tempo necessario agli operatori per mandare in esecuzione i compiti uno dopo l'altro, e di conseguenza aumentò il carico di lavoro che i computer potevano sostenere.

Man mano che la potenza dei computer cresceva, risultò chiaro che l'elaborazione batch monoutente non consentiva di sfruttare pienamente le loro potenzialità. Si pensò allora che i vari processi potessero *condividere* (in inglese *share*) le risorse del computer per utilizzarlo al meglio. Questa idea è conosciuta con il nome di *multiprogrammazione*: essa prevede l'elaborazione simultanea di diversi compiti sullo stesso computer. Le risorse vengono condivise tra i diversi compiti, man mano che ciascuno di essi ne fa richiesta. Nei primi sistemi multiprogrammati gli utenti inserivano ancora i compiti con le vecchie schede perforate e aspettavano anche diversi giorni per ottenere i risultati.

Negli anni '60 alcuni gruppi industriali e alcune Università sperimentarono una nuova categoria di sistemi operativi, detti di tipo *timesharing* (in inglese "condivisione di tempo"). Il timesharing è un caso particolare di sistema multiprogrammato, in cui gli utenti accedono al computer attraverso dei *terminali* (dispositivi tipicamente composti da uno schermo e una tastiera). In un tipico sistema timesharing possono esserci decine o anche centinaia di utenti che condividono nello stesso momento le risorse del computer centrale. Il computer non soddisfa simultaneamente tutte le richieste degli utenti, ma esegue una piccola parte del compito di un utente, quindi passa a eseguire una piccola parte del compito di un altro utente e così via. La velocità di esecuzione tuttavia è talmente elevata che il computer può tornare sulla richiesta di un dato utente diverse volte in un secondo. Per questo motivo i programmi degli utenti *sembrano* essere eseguiti tutti simultaneamente. Un vantaggio del timesharing consiste nel fatto che la richiesta di un utente viene soddisfatta quasi istantaneamente, anziché richiedere diverse ore come accadeva in precedenza.

1.5 I personal computer, i sistemi distribuiti e i sistemi client/server

Nel 1977 la Apple Computer costruì e rese popolari una nuova categoria di computer a basso costo: i *personal computer*. Mentre agli albori dell'informatica possedere un calcolatore era il sogno di una minoranza di appassionati, da allora i computer divennero abbastanza economici per essere acquistati da aziende e privati. Nel 1981 IBM, leader mondiale nel campo dell'hardware, commercializzò il Personal Computer IBM. Nel giro di poco tempo il personal computer divenne onnipresente in aziende, industrie e organizzazioni governative.

Questi computer, tuttavia, erano delle entità singole e separate, per cui per condividere le proprie informazioni con gli altri, dopo aver lavorato su una macchina, era necessario trasportarle sui dischetti. Ben presto ci fu la possibilità di collegare in rete tutti i computer di un'organizzazione (tramite la linea telefonica o reti *LAN*, cioè reti locali), il che diede origine ai cosiddetti *sistemi distribuiti*: le elaborazioni di un'organizzazione non erano più legate ad una postazione fissa, ma potevano essere distribuite su una rete. Su ogni nodo della rete è presente un computer dove si effettua una parte di lavoro, secondo la competenza specifica di quella divisione all'interno dell'organizzazione.

Soltanto una decina di anni fa i personal computer potenti come quelli odierni potevano costare milioni di dollari. Le macchine più potenti dell'attuale generazione di calcolatori (dette *workstation*) offrono delle prestazioni veramente incredibili. Le informazioni sono istradate su reti telematiche ad alta velocità, dove alcuni computer detti *file server* conservano le informazioni comuni: a esse accedono tanti computer detti *client* sparsi su tutta la rete. Tali sistemi di elaborazione prendono appunto il nome di sistemi *client/server*. I sistemi operativi più utilizzati oggi, come UNIX, Linux e Microsoft Windows offrono comunemente queste funzionalità. Il C e il C++ sono diventati i linguaggi di elezione per scrivere applicazioni quali sistemi operativi, sistemi di rete e sistemi distribuiti di tipo client/server.

1.6 I linguaggi macchina, assembly e ad alto livello

I programmatori possono scrivere le istruzioni da fare eseguire ad un computer in vari linguaggi di programmazione. Alcuni linguaggi sono interpretati direttamente dai computer, altri richiedono dei passi intermedi di *traduzione*. Allo stato attuale contiamo centinaia di linguaggi di programmazione diversi che, però, possono essere catalogati in tre grandi categorie:

1. Linguaggi macchina
2. Linguaggi assembly
3. Linguaggi ad alto livello

Un computer è in grado di capire soltanto il proprio *linguaggio macchina*. Dobbiamo considerarlo un po' come la sua lingua naturale, in quanto è definito durante la sua progettazione hardware. In genere i linguaggi macchina consistono di stringhe numeriche (che si riducono infine a sequenze di 0 e 1) che ordinano al computer di eseguire sequenze di

operazioni elementari. I linguaggi macchina *dipendono* dall'architettura del computer su cui sono definiti, e sono assolutamente illeggibili agli occhi di un essere umano. Se volete una prova eccovi un frammento di un programma in linguaggio macchina che somma lo straordinario alla paga base ed ottiene come risultato della somma la paga lorda:

```
+1300042774
+1400593419
+1200274027
```

Con l'accresciuta popolarità dei computer ci si rese conto che continuare a programmare in linguaggio macchina era un'operazione decisamente noiosa, lenta e soggetta ad errori. Anziché scrivere stringhe numeriche che i computer potevano capire direttamente, i programmatori cominciarono a far uso di abbreviazioni in lingua inglese, acronimi delle operazioni elementari. Tali abbreviazioni costituirono la base dei *linguaggi assembly*. Parallelamente furono sviluppati dei *programmi di traduzione* detti *assembler* che convertivano i programmi scritti in linguaggio assembly in linguaggio macchina. Guardate come appare la stessa sezione del programma del salario in linguaggio assembly:

```
LOAD   PAGABASE
ADD    STRAORDINARIO
STORE  PAGALORDA
```

Questo codice è più chiaro per un essere umano ma, per contro, non è di nessun significato per un computer finché non viene tradotto nel suo linguaggio macchina.

L'utilizzo dei computer aumentò sensibilmente dall'invenzione dei linguaggi assembly, ma in definitiva essi richiedevano ugualmente un gran numero di istruzioni per eseguire il compito più banale. Per velocizzare la programmazione furono allora inventati i *linguaggi ad alto livello*, nei quali una singola istruzione poteva eseguire compiti complessi. Programmi di traduzione detti *compilatori* convertivano poi i programmi scritti nel linguaggio ad alto livello in linguaggio macchina. I linguaggi ad alto livello consentivano di scrivere delle istruzioni che suonavano all'incirca come una frase inglese ordinaria e che contenevano comuni operatori matematici. Ecco la versione del programma di salario in un ipotetico linguaggio ad alto livello:

```
pagaLorda = pagaBase + Straordinario
```

Come intuite i linguaggi ad alto livello sono molto più vicini al modo di esprimersi di una persona dei linguaggi assembly o macchina. Il C ed il C++ sono tra i linguaggi ad alto livello più utilizzati al giorno d'oggi.

La compilazione di un programma ad alto livello in linguaggio macchina può comunque richiedere un discreto tempo di elaborazione. Per ovviare a questo inconveniente furono inventati i cosiddetti *interpreti*, in grado di eseguire direttamente i programmi scritti in linguaggio ad alto livello senza doverli prima convertire in linguaggio macchina. Anche se i programmi compilati sono più veloci di quelli interpretati, gli interpreti sono molto utilizzati in ambienti di sviluppo in cui è necessario apportare continue modifiche ai programmi: con gli interpreti si evita, infatti, di effettuare frequentemente l'operazione di compilazione. Una volta raggiunta la versione definitiva di un programma, però, è possibile creare la sua versione compilata per ottenerne una versione più efficiente.

1.7 Il C e il C++: un po' di storia

Il C++ è l'evoluzione naturale del C, il quale a sua volta discende da due linguaggi, il BCPL e il B. Il BCPL fu sviluppato nel 1967 da Martin Richards per scrivere software dei sistemi operativi e compilatori. Ken Thompson modellò molte caratteristiche del suo linguaggio B sulla base del BCPL, e utilizzò il B per creare le prime versioni del sistema operativo UNIX presso i Bell Laboratories nel 1970 su un computer DEC PDP-7. Sia il B che il BCPL erano linguaggi che non prevedevano *tipi di dato*: ogni dato occupava una *parola* di memoria e la responsabilità di vedere un dato come numero intero o numero reale ricadeva tutta sul programmatore.

Il linguaggio C si sviluppò dal B grazie a Dennis Ritchie presso i Bell Laboratories, e fu implementato originariamente su un computer DEC PDP-11 nel 1972. Il C riprende molti concetti utili del B e del BCPL, e in più aggiunge il concetto fondamentale di tipo di dato. Il C fu noto inizialmente come linguaggio di sviluppo del sistema operativo UNIX. Oggi la maggior parte dei sistemi operativi sono scritti in C e/o in C++. Nei due decenni alle nostre spalle il C è diventato disponibile sulla maggior parte dei computer ed è indipendente dall'hardware. Con una progettazione attenta potete scrivere programmi in C che possono girare praticamente su qualsiasi tipo di computer.

Alla fine degli anni '70 il C si è evoluto in quello che ora chiamiamo il "C tradizionale", il "C classico" o il "C di Kernighan e Ritchie"

L'utilizzo del C su diversi tipi di computer (che possiamo anche chiamare *piattaforme hardware*) ha portato purtroppo anche alla circolazione di molte sue varianti. La maggior parte di esse sono simili, ma spesso anche incompatibili. Ciò era visto come un problema dai programmatori, che avevano bisogno di scrivere software portabile su parecchie piattaforme. A un certo punto si avvertì la necessità di una versione standard del linguaggio. Nel 1983 vide la luce la commissione tecnica X3J11, che faceva capo alla commissione americana per gli standard X3, con l'obiettivo di stabilire una definizione del linguaggio chiara e indipendente dall'hardware. Lo standard fu approvato nel 1989: l'organizzazione ANSI collaborò con l'organizzazione internazionale per gli standard (ISO) per standardizzare il C in tutto il mondo. Il documento congiunto per la definizione dello standard fu pubblicato nel 1990 ed è noto come ANSI/ISO 9899:1990. Potete ordinare delle copie di questo documento direttamente dall'ANSI. La seconda edizione del testo di Kernighan e Ritchie, del 1988, riflette questa versione detta ANSI C. questa è la versione del linguaggio utilizzata oggi in tutto il mondo.

Obiettivo portabilità 1.1

Il C è stato standardizzato, è indipendente dall'hardware ed è disponibile sulla maggior parte dei sistemi, perciò le applicazioni scritte in C possono essere spesso eseguite, con qualche o addirittura nessuna modifica, su gran parte dei sistemi esistenti.

Il C++, come estensione del C, è stato sviluppato da Bjarne Stroustrup agli inizi degli anni '80 presso i Bell Laboratories. Il C++ offre una serie di funzionalità che "abbelliscono" il C, e, cosa più importante, lo rendono un ottimo strumento per la *programmazione orientata agli oggetti*.

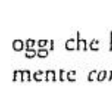
La comunità del software sta attraversando una rivoluzione. Scrivere software in modo veloce, corretto ed economico rimane uno degli obiettivi più importanti, specialmente

oggi che la domanda di software nuovo e potente è in crescita. Gli *oggetti* sono essenzialmente *componenti* software riutilizzabili, modellati sul mondo reale. Gli sviluppatori di software si rendono conto che una progettazione modulare e orientata agli oggetti può rendere i gruppi di programmazione molto più produttivi rispetto ai vecchi metodi, come la programmazione strutturata. I programmi orientati agli oggetti sono più semplici da leggere, da correggere e da modificare.

Sono stati sviluppati diversi altri linguaggi orientati agli oggetti, fra cui citiamo Smalltalk, sviluppato presso il PARC (Palo Alto Research Center) di Xerox. Smalltalk è un linguaggio "puro" orientato agli oggetti: in parole povere in esso esistono soltanto oggetti. Il C++ è un linguaggio ibrido: è possibile programmare in C++ nel vecchio stile C, nel nuovo stile orientato agli oggetti, o in entrambi. Nella Sezione 1.9 parleremo del nuovo linguaggio che si basa su C e C++: Java.

1.8 La libreria standard del C++

I programmi in C++ si compongono di *classi* e *funzioni*. Potete definire un numero arbitrario di queste componenti, a seconda delle vostre esigenze. Tuttavia, molti programmatori preferiscono affidarsi alla ricca collezione di classi e funzioni già esistenti nella libreria standard del C++. In fin dei conti le due cose che occorrono per poter programmare in modo efficace in C++ sono il linguaggio C++ stesso e un uso ottimale delle classi e delle funzioni della libreria standard. Nel corso del libro avremo modo di incontrarne un gran numero. In commercio troverete diversi testi che parlano più diffusamente delle funzioni di libreria dell'ANSI C (presenti anche nel C++). Le librerie standard delle classi sono generalmente fornite dai produttori del compilatore. Diversi rivenditori forniscono poi librerie di classi specializzate per altri scopi (ad es. per applicazioni scientifiche o per l'accesso a basi di dati).



Ingegneria del software 1.1

Il modo migliore per creare dei programmi è con l'approccio "a blocchi": è inutile inventare la ruota ogni volta. Dove è possibile utilizzate pezzi già esistenti. Questo principio è noto come riutilizzo del software ed è alla base della programmazione orientata agli oggetti.

Ingegneria del software 1.2

Programmando in C++ vi troverete a utilizzare generalmente le seguenti componenti: classi e funzioni della libreria standard del C++, classi e funzioni che avrete creato voi stessi e, infine, classi e funzioni di librerie non standard ma ampiamente diffuse.

Il vantaggio di creare funzioni e classi proprie sta nel fatto che si sa esattamente come funzionano e si può esaminare il loro codice sorgente. Gli svantaggi principali sono la quantità di tempo e lo sforzo che occorrono per progettarle, svilupparle e mantenerle.

Obiettivo efficienza 1.1

L'utilizzo di classi e funzioni della libreria standard al posto di codice scritto di vostro pugno può migliorare sensibilmente le prestazioni dei vostri programmi: la libreria standard è stata scritta ponendo particolare attenzione all'efficienza di esecuzione.

Obiettivo portabilità 1.2

L'utilizzo di classi e funzioni della libreria standard al posto di codice scritto di vostro pugno può aumentare la portabilità dei vostri programmi: la libreria standard è inclusa in tutte le implementazioni del C++.

1.9 Java, Internet e il World Wide Web

Nel Maggio 1995 la Sun Microsystem, uno dei leader nella produzione di workstation UNIX e di strumenti ed applicazioni di rete, annunciò lo sviluppo di un nuovo linguaggio di programmazione chiamato Java. Java è fondamentalmente simile al C ed al C++ e incorpora caratteristiche proprie di altri linguaggi orientati agli oggetti. La Sun fornisce gratuitamente il compilatore ed il software di sviluppo di Java, la documentazione, i tutorial ed alcuni programmi dimostrativi sul sito web www.javasoft.com. Java include librerie di classi specializzate per applicazioni multimediali, per applicazioni di rete, per il multithreading, la grafica, l'accesso ai database e la gestione dei sistemi distribuiti.

Una delle caratteristiche più importanti di Java è l'estrema portabilità: è possibile scrivere un programma in Java su un qualunque computer e eseguirlo su qualsiasi altro computer che supporti Java, non importa a quale piattaforma essi appartengano (la maggior parte dei sistemi attualmente è in grado di supportare il linguaggio). Questa caratteristica è molto gradita dagli sviluppatori di software, in special modo da coloro che fino ad ora sono stati costretti a scrivere versioni differenti dei propri programmi a seconda del computer sul quale sarebbero stati eseguiti. Ciò ha costituito, ovviamente, un notevole spreco di tempo e di denaro, ed inoltre ha scoraggiato finora i produttori di software dal produrre programmi per le piattaforme meno diffuse. Oggi le applicazioni Java possono essere eseguite su tutte le versioni più comuni di Windows e su altre piattaforme importanti, come le diverse varianti di UNIX, Apple Macintosh e IBM OS/2.

1.10 Altri linguaggi ad alto livello

Sono centinaia i linguaggi ad alto livello sviluppati fino ad oggi, ma soltanto pochi di essi hanno conosciuto una grande popolarità. Il FORTRAN (*FORmula TRANslator*) fu sviluppato dall'IBM tra il 1954 e il 1957 per applicazioni scientifiche e tecniche, che richiedessero calcoli matematici complessi. Il FORTRAN è ancora utilizzato, specialmente nell'ambiente scientifico.

Il COBOL (*Common Business Oriented Language*) fu sviluppato nel 1959 da costruttori di computer, industrie ed organismi governativi statunitensi. Il COBOL è utilizzato principalmente in applicazioni commerciali per le quali è necessaria la manipolazione efficiente di grandi quantità di dati. Ad oggi, più del 50% delle applicazioni commerciali sono scritte in COBOL.

Il Pascal fu progettato più o meno negli stessi anni del C dal professor Niklaus Wirth, soprattutto per usi accademici. Esso sarà presentato più diffusamente nella prossima sezione.

1.11 La programmazione strutturata

Nel corso degli anni '60 il mercato del software ha conosciuto una notevole crisi: le commesse di software venivano evase generalmente in ritardo, i costi eccedevano di gran lunga i budget previsti e i prodotti finiti non erano totalmente affidabili. Ci si accorse, dunque, che la creazione di software era un'attività decisamente più complessa di quanto si pensava. L'attività di ricerca di quegli anni portò all'evoluzione della *programmazione strutturata:* un approccio disciplinato alla scrittura del software, realizzata con lo scopo di rendere i programmi più chiari e più semplici da verificare e correggere. Il Capitolo 2 illustrerà i principi della programmazione strutturata. Nei Capitoli 3, 4 e 5 vedremo degli esempi di programmi strutturati.

Uno dei risultati più tangibili di questa ricerca fu lo sviluppo del linguaggio di programmazione Pascal, ad opera di Niklaus Wirth nel 1971. Il Pascal, denominato così in onore al matematico del XVII secolo Blaise Pascal, fu espressamente progettato per insegnare la programmazione strutturata negli ambienti accademici, e divenne rapidamente il linguaggio di programmazione più utilizzato in molte Università. Purtroppo il Pascal non possiede molte delle caratteristiche necessarie nelle applicazioni commerciali ed industriali, quindi non ha conosciuto una grossa popolarità al di fuori dell'ambiente di Ricerca.

Il linguaggio Ada fu sviluppato nell'ambito di un progetto finanziato dal Dipartimento di Difesa degli Stati Uniti (DoD) tra la fine degli anni '70 e l'inizio degli anni '80. I sistemi di controllo complessi presenti presso il DoD erano stati sviluppati in numerosi linguaggi di programmazione; il DoD, però, era alla ricerca di un solo linguaggio capace di soddisfare tutte le sue esigenze. Scegliendo come base alcune caratteristiche del linguaggio Pascal, si sviluppò un linguaggio chiamato Ada (che, nei fatti, risulta piuttosto differente dal Pascal), in onore di Lady Ada Lovelace, figlia del poeta Lord Byron. Lady Lovelace, agli inizi del XIX secolo, scrisse il primo "programma" per calcolatore conosciuto. Esso avrebbe dovuto funzionare su di un calcolatore meccanico progettato da Charles Babbage (l'*analytical engine*). Una caratteristica degna di nota del linguaggio Ada è il supporto al cosiddetto *multitasking*, cioè l'esecuzione parallela di diverse attività. Gli altri linguaggi ad alto livello di uso comune, compresi il C e il C++, consentono invece ai programmatori di eseguire una sola attività alla volta.

1.12 Gli elementi fondamentali di un tipico ambiente C++

Gli ambienti di sviluppo C++ generalmente sono formati da diverse componenti: un ambiente di sviluppo dei programmi, il linguaggio e la libreria standard del C++ (un tipico ambiente di sviluppo è illustrato in Figura 1.1).

Prima di giungere all'esecuzione, un programma C++ attraversa sei fasi (Figura 1.1): *scrittura/modifica, preprocessing, compilazione, linking (collegamento), caricamento* ed *esecuzione.* Nel seguito, ci riferiremo nelle nostre esemplificazioni ad un tipico sistema UNIX. In ogni caso, i programmi contenuti nel nostro libro potranno essere eseguiti, con nessuna o con minime modifiche, sulla maggior parte degli ambienti C++ esistenti compresi quelli disponibili sui sistemi operativi Microsoft. Se non state utilizzando un sistema UNIX fate riferimento ai manuali del vostro sistema o, eventualmente, chiedete al vostro sistemista di aiutarvi ad eseguire i programmi sul vostro sistema.

La prima fase è la *scrittura/modifica* di un file. Allo scopo si utilizza un programma detto *editor* che consente al programmatore di scrivere o modificare il suo programma. Il programma viene successivamente salvato su un'unità di memorizzazione secondaria, come per esempio il disco rigido, con la convenzione che i nomi dei file per i programmi C++ terminino con le estensioni `.cpp`, `.cxx` o `.C` (maiuscola). Consultate la documentazione del vostro ambiente di sviluppo per sapere che estensione utilizzare. Gli editor più comunemente utilizzati in ambiente UNIX sono `vi` e `emacs`. I pacchetti software come Borland C++ e Microsoft Visual C++ possiedono un proprio editor interno che si integra perfettamente con l'ambiente di programmazione. D'ora in avanti supporremo che sappiate utilizzare il vostro editor per scrivere i programmi.

Dopo avere scritto il programma è necessario *compilarlo*: il compilatore converte il codice del vostro programma C++ in linguaggio macchina (detto anche *codice oggetto*). In un ambiente C++ prima della fase di compilazione viene eseguita una pre-elaborazione del programma per mezzo di un componente software chiamato *preprocessore*. Il preprocessore C++ esegue delle istruzioni speciali dette *direttive al preprocessore*, le quali indicano alcune manipolazioni da effettuare sul codice prima della fase di compilazione. Tali manipolazioni consistono generalmente nell'inclusione di file esterni e nella sostituzione di stringhe all'interno del testo del programma. Nei primi capitoli introdurremo le più comuni direttive al preprocessore; in seguito, affronteremo organicamente l'argomento nel capitolo 6 del volume Tecniche Avanzate. È il compilatore che chiama il preprocessore prima di iniziare la compilazione.

La fase successiva è detta *collegamento* o *linking*. I programmi in C++ contengono spesso dei riferimenti a funzioni definite altrove, per esempio nelle librerie standard o in speciali librerie dedicate a particolari applicazioni. Il codice macchina prodotto dal compilatore (detto codice oggetto) contiene dunque dei riferimenti pendenti in corrispondenza di queste funzioni. Il *linker* lega il codice oggetto assieme con il codice delle funzioni mancanti per produrre quella che viene chiamata l'*immagine eseguibile* (o più semplicemente *l'eseguibile*), in cui non ci sono più riferimenti non risolti. Su un tipico sistema UNIX il comando per compilare ed effettuare il linking di un programma C++ è `CC`. Ad esempio, per compilare ed effettuare il linking del programma `welcome.C` digitate

```
CC welcome.C
```

al prompt dei comandi UNIX e premete il tasto *Invio*. Se la compilazione e il linking hanno successo verrà creato il file `a.out`; esso è l'eseguibile del nostro `welcome.C`.

La fase successiva è quella di *caricamento*: prima che un programma possa andare in esecuzione deve essere portato in memoria. E' il *loader* (caricatore) che prende l'immagine eseguibile e la trasferisce in memoria. Infine il computer, sotto il controllo della CPU, *esegue* il programma un'istruzione alla volta. Per caricare ed eseguire un programma in un sistema UNIX basta digitare `a.out` al prompt dei comandi e premere *Invio*.

Non sempre i programmi funzionano al primo tentativo: ciascuna delle fasi che abbiamo descritto può fallire a causa di diversi tipi di errori. Per esempio, un programma può tentare di dividere un numero per zero (operazione priva di significato in informatica come in matematica) e, in questo caso, il computer segnala un errore tramite un apposito messaggio. Il programmatore deve quindi ripercorrere tutte le fasi a partire da quella di scrittura/modifica per correggere l'errore e determinare se il programma modificato funzioni correttamente.

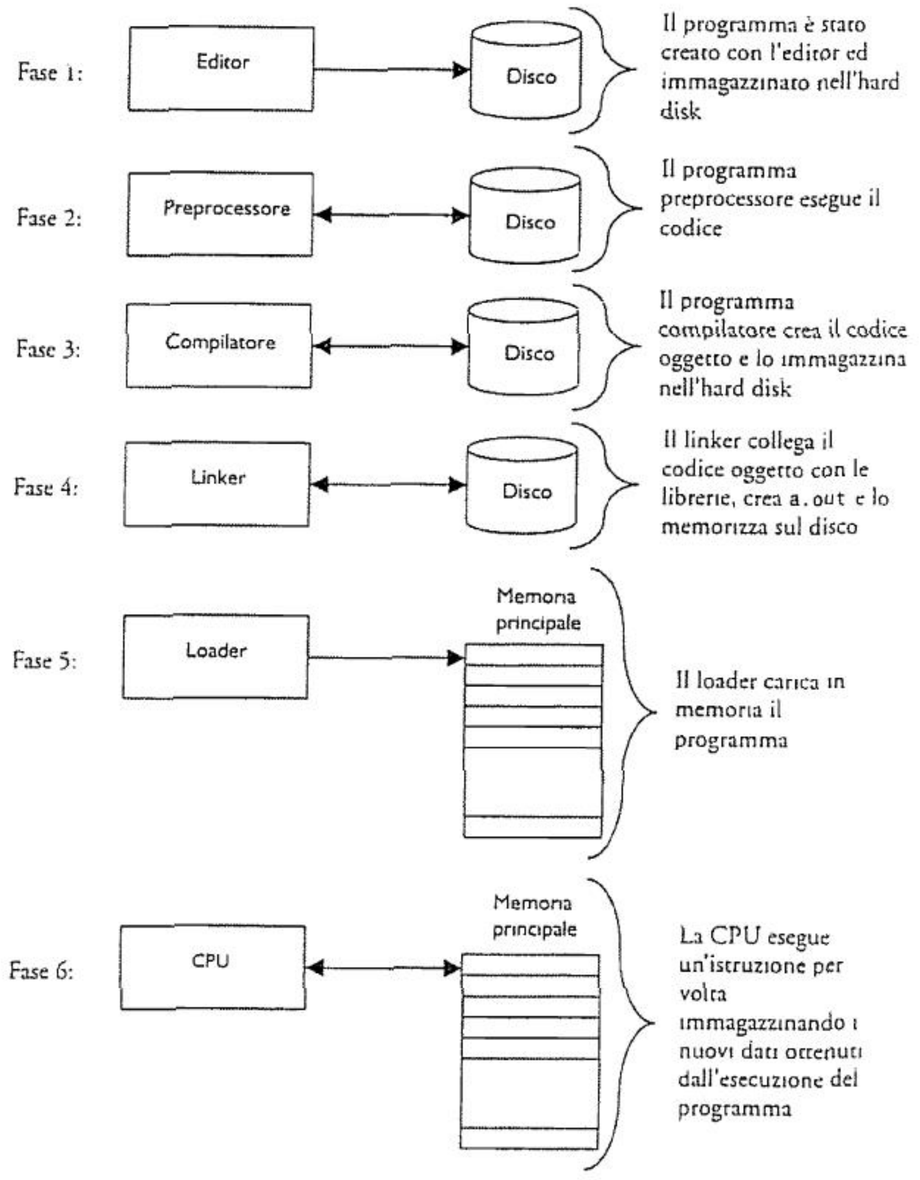


Figura 1.1 Un tipico ambiente di sviluppo C++

La maggior parte dei programmi riceve dei dati in input e ne invia altri in output. Alcune funzioni C++ prendono i dati in input dallo *stream* `cin` (nome del flusso di dati per l'input standard) che coincide normalmente con la tastiera; tuttavia `cin` può essere connesso ad altre unità. I dati sono quindi inviati in output sullo stream `cout` (flusso di dati per l'output standard) che coincide normalmente con lo schermo; allo stesso modo di `cin`, tuttavia, `cout` può essere connesso ad altre unità. Quando diciamo che un programma mostra un risultato, intendiamo generalmente che lo visualizza sullo schermo. I dati però

possono anche essere inviati in output su altri dispositivi, come i dischi o le stampanti. Esiste anche uno *stream di dati standard per gli errori* denominato `cerr`. Lo stream `cerr`, generalmente connesso allo schermo, viene utilizzato per visualizzare i messaggi di errore. Spesso i programmatori dirigono i dati destinati all'output, cioè a `cout`, a un'unità diversa dallo schermo, mantenendo l'associazione di `cerr` con lo schermo, in modo da essere informati immediatamente di eventuali errori.

Errore tipico 1.1

Errori come la divisione per zero si verificano in fase di esecuzione del programma, e sono detti perciò errori a tempo di esecuzione (in inglese run-time errors*). La divisione per zero è generalmente un errore fatale, cioè fa terminare immediatamente l'esecuzione del programma. Gli errori non fatali invece consentono al programma di completare il loro lavoro, anche se spesso i risultati non sono corretti. Nota: su alcuni sistemi la divisione per zero non è un errore fatale. Per sapere quale comportamento si ottiene in seguito a questo errore, consultate la documentazione del vostro sistema.*

1.13 Alcune considerazioni generali sul C++ e sul nostro corso

Il C++ è un linguaggio complesso. Alcuni programmatori esperti si vantano di scrivere il codice più contorto, oscuro e bizzarro possibile. Non si tratta ovviamente di una pratica molto conveniente: i programmi risultano poi difficili da leggere e spesso assumono comportamenti inspiegabili, per non parlare della difficoltà di correggere eventuali errori o effettuare qualche piccola modifica. Questo libro si rivolge ai principianti, per cui porremo sempre l'accento sulla *chiarezza*.

Buona abitudine 1.1

Scrivete il vostro codice C++ in modo semplice e diretto. In inglese questa pratica viene denominata KIS (keep it simple*: non complicatevi la vita!). Non forzate le caratteristiche del linguaggio per creare codice complesso e bizzarro.*

Nel testo includeremo diverse note di questo tipo per aiutarvi ad acquisire buone abitudini di programmazione, che vi consentano di scrivere codice chiaro, leggibile e semplice da mantenere e da correggere. Tutto ciò che possiamo darvi è qualche linea guida, con la pratica svilupperete senz'altro il vostro stile personale di programmazione. Evidenzieremo ogni tanto anche alcuni comuni errori di programmazione, alcuni problemi relativi alle prestazioni (per scrivere programmi più veloci e che utilizzano meno memoria), alcune questioni relative alla portabilità (perché i vostri programmi possano essere eseguiti sulla maggior parte di sistemi possibili), alcune problematiche relative allo sviluppo del software (per scrivere programmi ben strutturati) e, infine, alcuni problemi relativi alla verifica dei programmi (per individuare e correggere gli errori eventualmente presenti).

Abbiamo già detto che il C e il C++ sono linguaggi portabili e che i programmi scritti in questi linguaggi possono essere eseguiti su una moltitudine di sistemi differenti. *La portabilità però è spesso un obiettivo difficile da raggiungere.* Il documento dello standard ANSI C contiene una lunga lista di questioni relative alla portabilità.

Obiettivo portabilità 1.3

La diversità dei compilatori C e C++ e dei sistemi informatici pone una serie di problemi: la portabilità può diventare un obiettivo difficile da raggiungere. Il solo fatto di utilizzare il C/C++ non garantisce la portabilità di un programma: spesso occorre uno sforzo aggiuntivo per superare le differenze tra sistemi e tra compilatori.

Crediamo a questo punto di avervi presentato un excursus storico piuttosto completo del C++, dai suoi albori fino dello standard ANSI/ISO. Tenete presente però che ci sono delle sottigliezze e degli argomenti avanzati in cui non ci addentreremo. Se avete bisogno di dettagli tecnici del C++ vi suggeriamo di leggere il documento che definisce lo standard. Il documento è reperibile sul web all'indirizzo

`http://www.cygnus.com/misc/wp/`

Alla fine del testo, l'appendice D elenca diversi siti web sull'argomento.

Ci sono diverse caratteristiche del nuovo C++ non compatibili con le sue implementazioni meno recenti, per cui potrete trovare che alcuni programmi di questo libro non funzionano sui compilatori più datati.

Buona abitudine 1.2

Leggete i manuali relativi alla versione del C++ che state utilizzando. Riferitevi spesso ad essi per assicurarvi di utilizzare correttamente e al meglio le nuove caratteristiche del C++.

Buona abitudine 1.3

Computer e compilatore saranno i vostri migliori insegnanti. Se non vi sentite sicuri anche dopo aver letto i vostri manuali del C++, vi conviene fare dei piccoli esperimenti e vedere cosa accade. Impostate il compilatore in modo tale che vi segnali il maggior numero di avvertimenti (warning) e messaggi di errore. Potrete poi studiare ogni messaggio e correggere i vostri programmi di conseguenza.

Il C++ facilita la strutturazione e l'approccio disciplinato alla progettazione del software. Nelle sezioni che seguono introdurremo la programmazione in C++, presentando diversi esempi che illustrano le più importanti caratteristiche del linguaggio, ed analizzando ogni esempio riga per riga. Nel Capitolo 2 presenteremo organicamente la *programmazione strutturata* in C++. Continueremo ad utilizzare l'approccio strutturato fino al Capitolo 5, mentre dal Capitolo 6 in poi ci addentreremo nella programmazione orientata agli oggetti. Anche nei primi 5 capitoli faremo dei riferimenti alla programmazione orientata agli oggetti nelle sezioni intitolate "Pensare in termini di oggetti" Esse sono delle sezioni speciali in cui farete conoscenza con i concetti base relativi agli oggetti, e in cui presenteremo dei casi di studio che vi aiuteranno a pensare in termini di oggetti nei vostri esercizi di programmazione.

1.14 Un programma semplice: visualizzare una linea di testo

Il C++ utilizza una notazione che può sembrare perlomeno curiosa ai non addetti. Cominceremo a scrivere un programma che scrive una linea di testo. Il programma e il suo output sullo schermo sono mostrati in Figura 1.2.

```
1   // Fig. 1.2: fig01_02.cpp
2   // Un primo programma in C++
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "Welcome to C++!\n";
8
9      return 0;  // indica che il programma è terminato con successo
10  }
```

```
Welcome to C++!
```

Figura 1.2 Programma che visualizza una riga di testo.

Per quanto semplice, questo programma illustra molte caratteristiche importanti del C++. Prendiamo in considerazione ogni sua linea in dettaglio.

```
// Fig. 1.2: fig1_02.cpp
// Un primo programma in C++
```

Queste due linee iniziano entrambe con i caratteri `//` che indicano che il resto della linea è un *commento*. I programmatori inseriscono dei commenti per *documentare* i loro programmi, e quindi migliorarne la leggibilità. I commenti aiutano anche le persone che non hanno scritto il programma a leggerlo e capirlo. I commenti non hanno alcun effetto in fase di esecuzione: essi sono semplicemente ignorati dal compilatore, che non genera nessun codice oggetto relativo a essi. La seconda linea di commento descrive semplicemente lo scopo del programma. Un commento che inizia con i caratteri `//` è detto *commento su linea singola*, perché termina alla fine della linea corrente. È possibile utilizzare anche i commenti nello stile del C, questi iniziano con i caratteri `/*` e terminano con `*/` e possono occupare anche diverse linee.

Buona abitudine 1.4

Cominciate il vostro programma con una linea di commento che ne descriva lo scopo.

La linea

```
#include <iostream.h>
```

è una *direttiva al preprocessore*, cioè un comando dato al preprocessore C++. Le linee che iniziano con il carattere # sono elaborate dal preprocessore prima che il programma venga compilato. La linea in esame dice al preprocessore di includere nel programma il contenuto del file `iostream.h`, che è un *file di intestazione* che definisce gli stream di input/output. Questo file deve essere incluso in ogni programma che invia dei dati in output sullo schermo o li riceve in input dalla tastiera utilizzando gli stream C++. Il programma in

Figura 1.2 invia i dati in output sullo schermo, come vedremo presto. Parleremo più diffusamente in seguito del contenuto di `iostream.h`. Attenzione: lo standard del C++ più recente indica che `iostream.h` e i file di intestazione in generale devono comparire senza `.h`: si dovrebbe quindi scrivere semplicemente `iostream`. Per quanto ci riguarda continueremo a scrivere i file di intestazione alla vecchia maniera, anche perché la maggior parte dei compilatori non supporta ancora il nuovo stile. Nella sezione 1.20 riprenderemo questo primo esempio per vedere come appare invece nel nuovo stile.

Errore tipico 1.2

Se dimenticate di includere il file `iostream.h` in un programma che riceve dati in input dalla tastiera o li inoltra in output sullo schermo, il compilatore vi segnalerà un errore.

La linea

```
int main()
```

fa parte di tutti i programmi in C++. Le parentesi che seguono `main` indicano che si tratta di un blocco del programma detto *funzione*. I programmi contengono una o più funzioni, una delle quali deve necessariamente chiamarsi `main`. Il programma in Figura 1.2 contiene soltanto una funzione. L'esecuzione di un programma in C++ inizia dalla funzione `main`, anche se essa non è la prima ad apparire nel listato. La parola chiave `int` a sinistra di `main` indica che `main` restituisce un numero intero. Spiegheremo in seguito ciò che significa quando studieremo le funzioni, nel Capitolo 3. Per ora includete semplicemente la parola chiave `int` prima di `main` nei vostri programmi.

La *parentesi graffa aperta* `{` inizia il *corpo* di ogni funzione mentre la corrispondente *parentesi graffa chiusa* `}` lo termina. La linea

```
cout << "Welcome to C++!\n";
```

indica al computer di visualizzare sullo schermo la *stringa* di caratteri compresa tra la coppia di doppi apici. Tutta la linea, compreso `cout`, l'*operatore* `<<`, la *stringa* `"Welcome to C++!\n"`, e il *punto e virgola* (;) costituisce un'*istruzione*. Ogni istruzione deve terminare con un punto e virgola (detto perciò *terminatore di istruzione*). Le operazioni di input/output in C++ sono effettuate per mezzo di *stream* (flussi) di caratteri. In questo modo, quando viene eseguita l'istruzione precedente, viene inviato il flusso di caratteri `Welcome to C++!` verso l'*oggetto stream di output standard*, cioè `cout`, normalmente connesso allo schermo. Parleremo di `cout` più in dettaglio nel Capitolo 11.

L'operatore `<<` viene detto *operatore di inserimento nello stream*. Durante l'esecuzione del programma il valore a destra dell'operatore (detto *operando destro*) viene inserito nello stream di output. I caratteri dell'operando destro vengono generalmente visualizzati esattamente come appaiono fra le coppie di doppi apici. Notate però che i caratteri `\n` non vengono visualizzati sullo schermo. La barra rovesciata (\) è denominata *carattere di escape* e serve per indicare dei caratteri speciali. Quando si incontra una barra rovesciata in una stringa di caratteri, il carattere che segue la barra si combina a essa per formare una *sequenza di escape*. La sequenza di escape `\n` significa *nuova linea*. Come risultato il *cursore*, che indica la posizione corrente sullo schermo, si sposta all'inizio della linea successiva. Elenchiamo altre sequenze di escape in Figura 1.3.

Sequenza di escape	Descrizione
`\n`	Nuova linea. Posiziona il cursore dello schermo sulla riga successiva.
`\t`	Tabulazione orizzontale. Sposta il cursore sulla tabulazione successiva.
`\r`	Carattere "a capo" Posiziona il cursore all'inizio della linea corrente, senza avanzare di una riga.
`\a`	Campanello. Emette un suono (beep).
`\\`	Barra inversa. Utilizzata per visualizzare il carattere di barra rovesciata.
`\"`	Apice doppio. Utilizzata per visualizzare il carattere di apice doppio.

Figura 1.3 Alcune comuni sequenze di escape.

Errore tipico 1.3

Se si omette il punto e virgola al termine di un'istruzione si commette un errore di sintassi. Ciò significa che il compilatore non riesce a riconoscere un'istruzione. Normalmente il compilatore visualizza anche un messaggio per aiutare il programmatore a localizzare e correggere l'errore. Gli errori di sintassi sono violazioni delle regole del linguaggio. Essi vengono anche detti errori di compilazione o errori in fase di compilazione, perché compaiono per l'appunto durante la compilazione.

La linea

```
return 0;  // indica che il programma è terminato con successo
```

si trova al termine di ogni funzione `main`. La parola chiave `return` costituisce uno dei tanti mezzi per *uscire da una funzione*. Nel nostro caso, usata al termine di `main`, il valore 0 indica che il programma è terminato con successo. Nel Capitolo 3 parleremo più a fondo delle funzioni e chiariremo le ragioni per cui inserire questa istruzione. Per ora includetela nei vostri programmi: se la omettete, su alcuni sistemi, potreste avere un avvertimento del compilatore.

La parentesi graffa chiusa } indica la fine di `main`.

Buona abitudine 1.5

Molti programmatori terminano la visualizzazione di una linea di testo con la sequenza nuova linea (`\n`). Ciò assicura che il cursore si trovi automaticamente su una nuova linea dello schermo. Convenzioni di questo genere sono utili se si vuole sfruttare il riutilizzo del software, uno degli obiettivi chiave dei progetti complessi.

Buona abitudine 1.6

Conviene rientrare l'intero corpo della funzione di un dato numero di spazi rispetto alle parentesi graffe che lo delimitano. In questo modo si migliora la leggibilità della funzione.

Buona abitudine 1.7

Decidete una convenzione per il rientro da applicare e utilizzatela in modo uniforme. Potete utilizzare i caratteri di tabulazione per rientrare il testo, ma spesso le tabulazioni hanno lunghezza non fissa. Noi vi consigliamo di utilizzare 3 spazi per ogni livello di rientro.

`Welcome to C++!` può essere visualizzato in diversi modi. Per esempio il programma in Figura 1.4 utilizza più di un'istruzione con gli operatori di inserimento nello stream: il risultato del programma è comunque identico a quello di Figura 1.2. Ciò accade perché ogni istruzione di inserimento nello stream riprende la visualizzazione esattamente nel punto dove l'istruzione precedente l'ha terminata. La prima istruzione visualizza `Welcome` e uno spazio, mentre la seconda comincia la visualizzazione sulla stessa linea subito dopo lo spazio. In generale il C++ consente ai programmatori di esprimere le istruzioni in diversi modi.

Una singola istruzione può visualizzare anche molte linee di testo se si utilizzano i caratteri di nuova linea, come in Figura 1.5. Per ogni sequenza `\n` il cursore viene posizionato all'inizio della linea successiva dello schermo. Se volete visualizzare una linea vuota basta che poniate due sequenze di escape nuova linea consecutivamente, come in Figura 1.5.

```
1   // Fig. 1.4: fig01_04.cpp
2   // Visualizzazione di una linea con più istruzioni
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "Welcome ";
8      cout << "to C++!\n";
9
10     return 0;  // indica che il programma è terminato con successo
11  }
```

```
Welcome to C++!
```

Figura 1.4 Visualizzazione di una riga utilizzando due istruzioni cout distinte.

```
1   // Fig. 1.5: fig01_05.cpp
2   // Visualizzazione di più linee con una sola istruzione
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "Welcome\nto\n\nC++!\n";
8
9      return 0;  // indica che il programma è terminato con successo
10  }
```

Figura 1.5 Visualizzazione di più righe utilizzando una sola istruzione cout (continua)

```
Welcome
to

C++!
```

Figura 1.5 Visualizzazione di più righe utilizzando una sola istruzione cout.

1.15 Un altro semplice programma: l'addizione di due numeri interi

Il nostro prossimo esempio utilizza lo stream di input `cin` e l'*operatore di estrazione dallo stream*, `>>`, per ricevere due numeri interi digitati dall'utente alla tastiera; successivamente il programma somma questi due valori e invia il risultato in output tramite `cout`. Il programma e un esempio di output sono illustrati in Figura 1.6.

```
1   // Fig. 1.6: fig01_06.cpp
2   // Programma di addizione
3   #include <iostream.h>
4
5   int main()
6   {
7      int integer1, integer2, sum;      // dichiarazione
8
9      cout << "Enter first integer\n";  // prompt
10     cin >> integer1;                  // legge un intero
11     cout << "Enter second integer\n"; // prompt
12     cin >> integer2;                  // legge un intero
13     sum = integer1 + integer2;        // assegnamento di sum
14     cout << "Sum is " << sum << endl; // visualizza sum
15
16     return 0;  // indica che il programma è terminato con successo
17  }
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Figura 1.6 Programma di addizione.

I commenti

```
// Fig. 1.6: fig1_06.cpp
// Programma di addizione
```

indicano il nome del file e lo scopo del programma. La direttiva al preprocessore

```
#include <iostream.h>
```

include il contenuto del file di intestazione `iostream.h` nel programma.

Come abbiamo già detto ogni programma inizia con la funzione `main`. Il corpo di questa funzione è delimitato da una coppia di parentesi graffe. La linea

```
int integer1, integer2, sum;      // dichiarazione
```

è una *dichiarazione*. Le parole `integer1`, `integer2` e `sum` sono nomi di *variabili*. Una variabile è una locazione nella memoria del computer dove può essere memorizzato un valore, in modo da poter essere utilizzato dal programma. La nostra dichiarazione specifica che le variabili `integer1`, `integer2` e `sum` sono di tipo `int`, il che significa che queste variabili conterranno valori di tipo *intero*. Per intero intendiamo un numero senza parte decimale come 7, -11, 0 e 31.914. Tutte le variabili devono essere dichiarate specificando il loro nome e il tipo di dati che conterranno prima di poter essere utilizzate nel programma. Più variabili dello stesso tipo si possono indicare in una sola o in più dichiarazioni. Avremmo potuto scrivere tre dichiarazioni per le nostre tre variabili, ma la soluzione che abbiamo adottato è certamente più concisa.

Buona abitudine 1.8

Alcuni programmatori preferiscono dichiarare ciascuna variabile su una linea diversa. Questa convenzione consente di aggiungere un piccolo commento dopo ogni variabile che ne descriva il suo utilizzo.

Parleremo presto dei tipi di dato `float` (per i numeri reali, cioè numeri che hanno una parte decimale, a es. 3.4, 0.0, -11.19) e `char` (per i caratteri alfanumerici: una variabile di tipo `char` può contenere un solo carattere numerico, alfanumerico o speciale come `5` o `*`).

Buona abitudine 1.9

Nelle dichiarazioni conviene digitare uno spazio dopo ogni virgola, per una migliore leggibilità.

Un nome di variabile viene anche detto *identificatore*. Esso è formato da una serie di caratteri che possono essere lettere, numeri e caratteri di sottolineatura (_); l'unica restrizione è che un identificatore non possa iniziare con un carattere numerico (ad esempio `1perTutti` non è un identificatore valido). Il C++, inoltre, fa distinzione fra lettere minuscole e maiuscole: le lettere minuscole sono interpretate come lettere diverse dalle corrispondenti maiuscole, per cui `a1` e `A1` sono due identificatori distinti.

Obiettivo portabilità 1.4

Il C++ consente l'utilizzo di identificatori di qualsiasi lunghezza, ma ci sono sistemi e implementazioni in cui è stabilita una lunghezza massima. Per assicurare la portabilità dei programmi vi conviene utilizzare identificatori non più lunghi di 31 caratteri.

Buona abitudine 1.10

Scegliere nomi di identificatori che hanno un significato evidente corrisponde a scrivere del codice "autodocumentante": per comprendere ciò che fa il programma non è necessario commentarlo eccessivamente.

Buona abitudine 1.11

Evitate identificatori che iniziano con i caratteri di sottolineatura perché generalmente i compilatori utilizzano identificatori di questo genere nei loro meccanismi interni di traduzione. In questo modo eviterete pericolose confusioni fra i vostri identificatori e quelli eventualmente generati dal compilatore.

Le dichiarazioni di variabili possono essere poste ovunque all'interno di una funzione. L'unica restrizione è che la dichiarazione compaia prima che la variabile sia effettivamente utilizzata nel programma. Nel programma di Figura 1.6, per esempio, avremmo potuto utilizzare tre dichiarazioni distinte per le tre variabili. La dichiarazione

```
int integer1;
```

sarebbe potuta comparire immediatamente prima della linea

```
cin >> integer1;
```

la dichiarazione

```
int integer2;
```

sarebbe potuta comparire immediatamente prima della linea

```
cin >> integer2;
```

e la dichiarazione

```
int sum;
```

sarebbe potuta comparire immediatamente prima della linea

```
sum = integer1 + integer2;
```

Buona abitudine 1.12

Lasciate sempre una linea vuota prima di una dichiarazione che compare tra due istruzioni: ciò rende le dichiarazioni più visibili, migliorando la leggibilità del programma.

Buona abitudine 1.13

Se preferite porre le dichiarazioni all'inizio delle funzioni, separatele dalle istruzioni con una o più linee vuote, per indicare chiaramente la fine delle dichiarazioni e l'inizio delle istruzioni eseguibili.

L'istruzione

```
cout << "Enter first integer\n";     // prompt
```

visualizza sullo schermo la stringa `Enter first integer` (detta anche *literal*) e posiziona il cursore all'inizio della riga successiva. Questo messaggio è detto *prompt* perché chiede all'utente di intraprendere un'azione specifica. Possiamo leggere questa istruzione in questo modo: `cout` riceve la stringa di caratteri `"Enter first integer\n"`

L'istruzione

```
cin >> integer1;      // legge un intero
```

utilizza l'*oggetto stream di input* `cin` e l'*operatore di estrazione dallo stream*, `>>`, per ricevere un valore dalla tastiera. Utilizzando l'operatore di estrazione dallo stream, `cin` riceve l'input dal flusso di input standard, che coincide normalmente con la tastiera. Possiamo leggere questa istruzione in questo modo: `cin` attribuisce un valore a `integer1`.

Quando viene eseguita l'istruzione, il computer attende che l'utente immetta un valore da assegnare a `integer1`. L'utente digita il valore e poi preme il tasto *Invio* per inviare il valore al computer. Esso viene poi assegnato a `integer1`. Tutti i riferimenti successivi a `integer1` nel programma utilizzeranno questo valore.

Gli oggetti stream `cin` e `cout` consentono l'interazione fra l'utente e il computer. Poiché questo tipo di interazione ricorda un dialogo, essa viene anche detta *elaborazione interattiva*.

L'istruzione

```
cout << "Enter second integer\n";     // prompt
```

visualizza le parole `Enter second integer` sullo schermo, posizionando poi il cursore all'inizio della riga successiva. L'istruzione

```
cin >> integer2;        // legge un intero
```

riceve un valore per la variabile `integer2` dall'utente.

L'istruzione di assegnamento

```
sum = integer1 + integer2;        // assegnamento di sum
```

calcola la somma delle variabili `integer1` e `integer2` e assegna il risultato alla variabile `sum`, utilizzando l'*operatore di assegnamento* `=`. La maggior parte dei calcoli viene effettuata nelle istruzioni di assegnamento. Gli operatori `=` e `+` sono detti *operatori binari* perché ognuno di essi ha due *operandi*. Nel caso dell'operatore `+` essi sono `integer1` e `integer2`. Nel caso dell'operatore `=` invece i due operandi sono `sum` e il valore dell'espressione `integer1 + integer2`.

Buona abitudine 1.14

Ponete degli spazi prima e dopo un operatore binario. In questo modo l'operatore sarà più visibile, migliorando la leggibilità del programma.

L'istruzione

```
cout << "Sum is " << sum << endl;      // visualizza sum
```

visualizza la stringa di caratteri `Sum is`, poi il valore numerico della variabile `sum`, e infine il *manipolatore di stream* `endl` ("end line", ovvero fine di linea). Il manipolatore `endl` invia in output un carattere di *nuova linea* (corrispondente al carattere `\n` visto in precedenza) e, successivamente, *svuota il buffer di output*. Su alcuni sistemi, infatti, l'output si accumula all'interno della macchina finché non ce n'è abbastanza perché sia il caso di visualizzarlo sullo schermo: in questi casi `endl` forza tutto l'output accumulato ad essere visualizzato in quel momento.

Notate come l'istruzione precedente invii in output valori di tipi diversi. L'operatore di inserimento nello stream "*sa*" come inviare in output ciascun tipo di dato. Se si utilizzano più operatori `<<` in una sola istruzione si dice anche che essi sono *concatenati*: non è necessario scrivere più linee per l'output di diversi valori.

È possibile addirittura effettuare i calcoli all'interno dell'istruzione di output. In questo caso abbiamo una combinazione delle istruzioni precedenti, del tipo:

```
cout << "Sum is " << integer1 + integer2 << endl;
```

si noti che la variabile `sum` non è più necessaria.

Subito dopo troviamo la parentesi graffa chiusa } che informa il computer della fine della funzione `main`.

Una caratteristica importante del C++ è che gli utenti possono creare tipi di dato personalizzati (lo vedremo nel Capitolo 6). Inoltre possono anche "*insegnare*" al C++ come gestire l'input e l'output dei loro nuovi tipi di dato tramite gli operatori << e >> (lo vedremo quando parleremo degli operatori sovraccarichi, nel Capitolo 8).

1.16 La memoria: concetti fondamentali

I nomi di variabili come `integer1`, `integer2` e `sum` corrispondono a *locazioni* specifiche nella memoria del computer. Ogni variabile ha un *nome*, un *tipo*, una *dimensione* e un *valore*.

Nel programma successivo in Figura 1.6, quando viene eseguita l'istruzione

```
cin >> integer1;
```

il valore digitato dall'utente viene posto nella locazione di memoria del computer che il compilatore ha predisposto per la variabile `integer1`. Supponiamo che l'utente digiti il valore `45`. Il computer porrà tale valore nella locazione di `integer1`, come mostra la Figura 1.7.

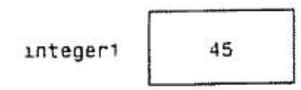


Figura 1.7 Locazione di memoria che mostra il nome e il valore di una variabile.

Quando si pone un valore in una locazione di memoria, esso rimpiazza il valore che vi era contenuto in precedenza. Tale valore viene irrimediabilmente perduto.

Tornando al nostro programma, supponiamo che quando viene eseguita l'istruzione

```
cin >> integer2;
```

l'utente immetta il valore `72`. Il computer porrà tale valore nella locazione relativa a `integer2` e la memoria apparirà come in Figura 1.8. Notate come non sia necessario che le due locazioni siano adiacenti.

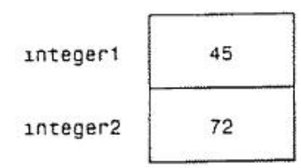


Figura 1.8 Locazioni di memoria dopo l'input dei valori di due variabili.

Una volta ottenuti i valori per `integer1` e `integer2`, essi sono sommati e il risultato è posto nella variabile `sum`. L'istruzione

```
sum = integer1 + integer2;
```

che effettua l'addizione distrugge anche un valore: stiamo parlando del valore che si trovava in precedenza nella locazione di `sum`. Dopo il calcolo di `sum` la memoria appare come in Figura 1.9. L'istruzione non modifica in alcun modo `integer1` e `integer2`. I loro valori

sono stati utilizzati ma non distrutti: la lettura di un valore di memoria non è un processo distruttivo.

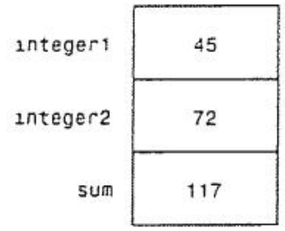


Figura 1.9 Locazioni di memoria dopo un calcolo.

1.17 I calcoli aritmetici

La maggior parte dei programmi effettua calcoli aritmetici. Gli *operatori aritmetici* sono riepilogati in Figura 1.10. Notate l'uso di vari simboli speciali diversi da quelli utilizzati in algebra. L'*asterisco* (`*`) indica la moltiplicazione, mentre il *simbolo percentuale* (`%`) è l'*operatore modulo*, che descriveremo fra breve. Gli operatori di Figura 1.10 sono tutti binari, cioè ognuno di essi prende due operandi.

Operazione C++	Operatore aritmetico	Espressione algebrica	Espressione C++
Addizione	+	$f + 7$	`f + 7`
Sottrazione	-	$p - c$	`p - c`
Moltiplicazione	*	bm	`b * m`
Divisione	/	x / y *oppure* $\frac{x}{y}$	`x / y`
Modulo	%	$r \bmod s$	`r % s`

Figura 1.10 Operatori aritmetici.

La *divisione intera* (con numeratore e denominatore interi) restituisce un valore intero: per es. l'espressione `7 / 4` restituisce `1`, e `17 / 5` restituisce `3`. In questi casi le parti decimali sono semplicemente scartate: si ha, cioè, un *troncamento* e non un *arrotondamento*.

In C++ è presente anche l'operatore *modulo*, `%`, che restituisce il resto di un'operazione di divisione intera. Possiamo utilizzarlo soltanto con operandi interi. L'espressione `x % y` restituisce il resto ottenuto dopo la divisione di `x` per `y`. Così `7 % 4` restituisce `3` e `17 % 5` restituisce `2`. L'operatore modulo ha molti usi interessanti, per esempio nel determinare se un numero è multiplo di un altro, oppure se è pari o dispari (lo vedremo in seguito).

Errore tipico 1.4

L'utilizzo di `%` con operandi non interi vi darà un errore di sintassi.

Le espressioni aritmetiche in C++ devono essere immesse su una sola linea: intendiamo che, per esempio, "a diviso b" deve apparire come `a / b`, in modo tale che costanti, variabili e operatori appaiano tutti in linea. La notazione algebrica

$$\frac{A}{B}$$

non è, in generale, accettabile anche se sono in circolazione degli speciali pacchetti software che supportano notazioni matematiche complesse.

Le parentesi sono utilizzate in C++ allo stesso modo in cui le usereste in un'espressione algebrica normale. Per esempio, per moltiplicare `a` per la quantità `b + c` si può scrivere:

```
a * (b + c)
```

Il C++ prende in considerazione gli operatori presenti in un'espressione algebrica in un ordine preciso, secondo la *precedenza* di ciascun operatore. Vi sono molti punti in comune fra le usuali precedenze degli operatori algebrici e quelle del linguaggio C++:

1. Gli operatori di espressioni contenute tra parentesi sono presi in considerazione per primi; potete cioè utilizzare le parentesi per *forzare* l'ordine di calcolo secondo le vostre esigenze. Le parentesi costituiscono l'*ordine massimo di precedenza*. In presenza di parentesi *nidificate*, viene presa in considerazione per prima l'espressione nelle parentesi più interne.
2. In seguito, vengono prese in considerazione le operazioni di moltiplicazione, di divisione e di modulo. Se in un'espressione ce ne sono parecchie, vengono calcolate così come si trovano, da sinistra a destra. Queste tre operazioni hanno lo stesso ordine di precedenza.
3. Per ultime troviamo le addizioni e le sottrazioni. Se in un'espressione ce ne sono parecchie, vengono calcolate così come si trovano, da sinistra a destra. Anche sottrazioni e addizioni hanno lo stesso ordine di precedenza.

Quando diciamo che gli operatori sono applicati da sinistra a destra, così come si trovano, ci riferiamo alla cosiddetta *associatività* degli operatori. Vedremo che alcuni operatori *associano* da destra a sinistra. In Figura 1.11 troviamo un riepilogo delle regole di precedenza. Espanderemo questa tabella man mano che incontreremo altri operatori. La lista completa è contenuta in appendice A.

Passiamo ora a considerare alcune espressioni, alla luce di quanto abbiamo detto sulle regole di precedenza. Ogni esempio mostra come tradurre un'espressione algebrica nella sua equivalente in C++.

Il primo esempio è una media aritmetica di 5 termini:

Algebra: $m = \frac{a + b + c + d + e}{5}$

```
C++: m = (a + b + c + d + e) / 5;
```

Le parentesi sono necessarie perché la divisione ha precedenza più alta dell'addizione. E' l'intera quantità `(a + b + c + d + e)` che deve essere divisa per `5`. Se omettiamo le parentesi, ciò che abbiamo è `a + b + c + d + e / 5`, che algebricamente significa un'altra cosa, cioè:

$$a + b + c + d + \frac{e}{5}$$

Il prossimo esempio illustra l'equazione di una *retta*:

Algebra: $y = mx + b$

```
C++: y = m * x + b;
```

Non c'è bisogno delle parentesi. Viene prima la moltiplicazione, perché ha precedenza più alta dell'addizione.

Operatore	Operazione	Ordine di valutazione (precedenza)
`( )`	Parentesi	Sono valutate per prime. Se le parentesi sono nidificate, viene valutata prima l'espressione dalla coppia di parentesi più interna. Se ci sono diverse coppie di parentesi "dello stesso livello", cioè non nidificate, sono valutate da sinistra a destra.
`*`, `/` oppure `%`	Moltiplicazione Divisione Modulo	Sono valutate per seconde. Se ce ne sono diverse, sono valutate da sinistra a destra.
`+` oppure `-`	Addizione Sottrazione	Sono valutate per ultime. Se ce ne sono diverse, sono valutate da sinistra a destra.

Figura 1.11 La precedenza degli operatori aritmetici.

L'esempio seguente contiene operazioni di modulo, moltiplicazione, divisione, addizione e sottrazione:

Algebra: $z = pr\%q + w/x - y$

```
C++: z = p * r % q + w / x - y;
       6   1   2   4   3   5
```

I numeri posti sotto l'istruzione indicano l'ordine in cui il C++ applica gli operatori. Vengono prima la moltiplicazione, il modulo e la divisione, da sinistra a destra (che è la loro direzione di associazione), perché hanno precedenza più alta di addizione e sottrazione. Quest ultime sono applicate in seguito, anch'esse da sinistra a destra.

Non tutte le parentesi multiple sono nidificate: nell'esempio seguente ci sono due coppie di parentesi *dello stesso livello*

```
a * (b + c) + c * (d + e)
```

Per comprendere anche meglio le regole di precedenza degli operatori consideriamo il calcolo del valore di un polinomio di secondo grado:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

I numeri posti sotto l'istruzione, indicano l'ordine in cui il C++ applica gli operatori. In C++ non disponiamo direttamente di un operatore *esponente*, per cui per calcolare il valore di `x` al quadrato abbiamo scritto `x*x`. Introdurremo in seguito la funzione di libreria `pow` (potenza), che effettua il calcolo della funzione elevamento a potenza; per comprenderla appieno, però, necessiteremo di alcuni chiarimenti relativi ai tipi di dato, per cui non potremmo discuterne prima del Capitolo 3.

Supponiamo che le variabili `a`, `b`, `c` e `x` siano state inizializzate ai seguenti valori: `a = 2`, `b = 3`, `c = 7` e `x = 5`. La Figura 1.12 illustra l'ordine in cui sono applicati gli operatori nel polinomio di secondo grado.

Per maggiore chiarezza possiamo anche utilizzare delle parentesi nell'istruzione di assegnamento, anche se non sono realmente necessarie:

```
y = (a * x * x) + (b * x) + c;
```

Passo 1. `y = 2 * 5 * 5 + 3 * 5 + 7;`

`2 * 5 =` (10) (Prima la moltiplicazione più a sinistra)

Passo 2. `y = 10 * 5 + 3 * 5 + 7;`

`10 * 5 =` (50) *(La moltiplicazione più a sinistra)*

Passo 3. `y = 50 + 3 * 5 + 7;`

`3 * 5 =` (15) (Prima la moltiplicazione dell'addizione)

Passo 4. `y = 50 + 15 + 7;`

`50 + 15 =` (65) (L'addizione più a sinistra)

Passo 5. `y = 65 + 7;`

`65 + 7 =` (72) (L'ultima addizione)

Passo 6. `y = 72` (L'ultima operazione: mettere 72 in y)

Figura 1.12 Ordine di valutazione di un polinomio di secondo grado.

Buona abitudine 1.15

Proprio come nelle espressioni aritmetiche, anche in C++ è possibile scrivere parentesi non necessarie per rendere le espressioni più chiare. Queste sono dette parentesi ridondanti. *Le parentesi ridondanti si utilizzano in genere per raggruppare le sottoespressioni presenti in una espressione estesa, rendendola più leggibile.*

1.18 Prendere decisioni: gli operatori relazionali e di uguaglianza

In questa sezione introduciamo una semplice versione del costrutto `if`, che ci consente di intraprendere un'azione sulla base della verità o falsità di una *condizione*. Se la condizione è soddisfatta, cioè è `true` (vera), viene eseguita l'istruzione che si trova nel corpo del costrutto `if`. Al contrario, in caso di condizione `false` (falsa), l'istruzione non verrà eseguita.

Le condizioni delle `if` possono essere formate con gli *operatori di uguaglianza* e gli *operatori relazionali*, riepilogati in Figura 1.13. Gli operatori relazionali hanno tutti lo stesso *livello di precedenza* e associano *da sinistra a destra*. *Anche gli operatori di uguaglianza* associano da sinistra a destra e hanno lo stesso livello di precedenza, che è però più basso di quello degli operatori relazionali.

Operatori algebrici standard relazionali e di uguaglianza	Operatori relazionali e di uguaglianza del C++	Esempio di condizione in C++	Significato della condizione in C++
Operatori di uguaglianza			
=	==	x == y	x è uguale a y
≠	!=	x != y	x *non* è uguale a y
Operatori relazionali			
>	>	x > y	x è maggiore di y
<	<	x < y	x è minore di y
≥	>=	x >= y	x è maggiore o uguale a y
≤	<=	x <= y	x è minore o uguale a y

Figura 1.13 Gli operatori relazionali e di uguaglianza.

Errore tipico 1.5

Se ponete uno spazio tra i due caratteri degli operatori ==, !=, >= e <= commettete un errore di sintassi.

Errore tipico 1.6

Se invertite l'ordine delle coppie di caratteri negli operatori !=, >= e <= commettete normalmente un errore di sintassi. In alcuni casi è possibile scrivere =! al posto di != , ma pur non commettendo un errore di sintassi si commette un errore logico.

Errore tipico 1.7

Non confondete l'operatore di assegnamento = con quello di uguaglianza == Il primo si legge "prende il valore di", il secondo "è uguale a". La confusione tra questi due operatori rappresenta uno degli errori più difficili da individuare in un programma.

L'esempio seguente utilizza sei istruzioni `if` per confrontare due numeri immessi dall'utente. Se la condizione di una qualsiasi delle `if` viene soddisfatta, viene eseguita l'istruzione di output di tale `if`. Il programma e tre esempi di output sono presentati in Figura 1.14.

Notate come il programma in Figura 1.14 usi operazioni di estrazione dallo stream a cascata per l'input dei due interi. Prima viene letto un valore nella variabile `num1` e poi un altro valore nella variabile `num2`. Il rientro delle istruzioni `if` migliora la leggibilità del programma. Inoltre notate come ogni istruzione `if` in Figura 1.14 ha una sola istruzione nel proprio corpo. Nel Capitolo 2 mostreremo come costruire delle `if` con più istruzioni al loro interno (includendole tutte tra un paio di parentesi graffe { }).

```
1  // Fig. 1.14: fig01_14.cpp
2  // Utilizzo di istruzioni if, di operatori
3  // relazionali e di operatori di uguaglianza
4  #include <iostream.h>
5
6  int main()
7  {
```

Figura 1.14 Utilizzo degli operatori relazionali e di uguaglianza (continua)

```
 8     int num1, num2;
 9
10     cout << "Enter two integers, and I will tell you\n"
11          << "the relationships they satisfy: ";
12     cin >> num1 >> num2;   // legge due interi
13
14     if ( num1 == num2 )
15        cout << num1 << " is equal to " << num2 << endl; // i due
16                                                 // numeri sono uguali
17     if ( num1 != num2 )
18        cout << num1 << " is not equal to " << num2 << endl; // i
19                                               // due numeri sono diversi
20     if ( num1 < num2 )
21        cout << num1 << " is less than " << num2 << endl; // il
22                                 // primo numero è minore dell'altro
23     if ( num1 > num2 )
24        cout << num1 << " is greater than " << num2 << endl; // il
25                                // primo numero è maggiore dell'altro
26     if ( num1 <= num2 )
27        cout << num1 << " is less than or equal to "  // il primo
28                               // numero è minore o uguale all'altro
29             << num2 << endl;
30
31     if ( num1 >= num2 )
32        cout << num1 << " is greater than or equal to " // il primo
33                                  // numero è maggiore o uguale all'altro
32             << num2 << endl;
33
34     return 0;  // indica che il programma è terminato con successo
35  }
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7

Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12

Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Figura 1.14 Utilizzo degli operatori relazionali e di uguaglianza.

Buona abitudine 1.16

Inserite un rientro prima dell'istruzione nel corpo di un costrutto `if` *sia per renderlo maggiormente visibile, che per migliorare la leggibilità del programma.*

Buona abitudine 1.17

Non dovreste porre più di un'istruzione per linea in un programma.

Errore tipico 1.8

Se mettete un punto e virgola subito dopo la parentesi destra della condizione di un costrutto `if` *comporta, il più delle volte, un errore logico (anche se non di sintassi). Il punto e virgola indica che l'istruzione contenuta nella* `if` *è l'istruzione vuota quindi, in pratica, non viene eseguito nulla indipendentemente dal fatto che la condizione sia vera o meno. Inoltre, l'istruzione destinata ad essere il corpo della* `if` *si trova ora di seguito dopo l'intero corpo della* `if`*, per cui verrà eseguita sempre e indipendentemente dalla* `if` *dando luogo ad un ulteriore comportamento errato.*

Notate l'utilizzo degli spazi in Figura 1.14. Normalmente il compilatore ignora gli spazi e i caratteri di nuova linea e di tabulazione. Quindi potete scegliere la spaziatura più consona alla vostre preferenze per le vostre istruzioni, ponendole anche su più righe. Attenzione però a non separare gli identificatori in più parti.

Errore tipico 1.9

Se inserite degli spazi all'interno di un identificatore commettete un errore di sintassi (per esempio se separate `main` *in* `ma in`*).*

Buona abitudine 1.18

Una istruzione lunga può essere suddivisa su più linee. Se lo fate scegliete con attenzione i punti in cui andare a capo, in modo possibilmente significativo: per esempio potete terminare una linea dopo una virgola, in una lista di elementi separati da virgole, oppure dopo un operatore, nel caso di una lunga espressione aritmetica. Rientrate le linee successive alla prima, in modo da accorgervi a colpo d'occhio che sono la continuazione di essa.

Lo schema in Figura 1.15 indica le precedenze degli operatori di cui abbiamo parlato in questo capitolo: essi sono elencati in ordine decrescente di precedenza. Con l'eccezione dell'operatore di assegnamento, `=`, tutti gli altri associano da sinistra a destra: quindi in un'addizione come `x + y + z` il calcolo procede come se fosse scritta `(x + y) + z`. Nel caso dell'espressione `x = y = 0`, essa viene calcolata come se fosse scritta `x = (y = 0)`, che come vedremo assegna prima `0` a `y`, quindi assegna il risultato di tale assegnamento (`0`) a `x`.

A questo punto avete fatto conoscenza con molte caratteristiche importanti del C++, fra cui la visualizzazione dei dati sullo schermo, l'input da tastiera, i calcoli matematici ed un modo per prendere decisioni. Nel Capitolo 2 approfondiremo queste tecniche, introducendo nello stesso tempo la *programmazione strutturata*. Utilizzeremo anche in modo più consapevole le tecniche di rientro. Inoltre, vedremo come specificare o variare l'ordine di esecuzione delle istruzioni ovvero, come specificare il *flusso di controllo*.

Operatori	Associatività	Tipo
( )	sx a dx	parentesi
* / %	sx a dx	moltiplicativo
+ -	sx a dx	additivo
<< >>	sx a dx	inserimento/estrazione su stream
< <= > >=	sx a dx	relazionale
== !=	sx a dx	di uguaglianza
=	dx a sx	assegnamento

Figura 1.15 Precedenza e associatività degli operatori discussi finora.

Buona abitudine 1.19

Se dovete scrivere un'espressione contenente molti operatori vi conviene far riferimento allo schema delle precedenze. In questo modo avrete la sicurezza che le operazioni si susseguono nell'ordine a cui stavate pensando. Se non ricordate perfettamente la precedenza di qualche operatore utilizzate le parentesi per forzare l'ordine di calcolo, esattamente come fareste in un'espressione algebrica. Ricordate che solamente pochi operatori, come quello di assegnamento (=), associano da destra a sinistra.

1.19 Le nuove convenzioni per i file di intestazione e gli spazi dei nomi

Questa sezione è dedicata ai programmatori che intendono utilizzare compilatori compatibili con lo standard ANSI/ISO. In esso sono specificati nuovi nomi per molti file di intestazione del C++, compreso `iostream.h`. La maggior parte dei nuovi file di intestazione non termina più con l'estensione `.h`. In Figura 1.16 abbiamo riscritto il programma presentato originariamente in Figura 1.2 in modo da renderlo conforme al nuovo formato per i file di intestazione e al modo in cui potete utilizzare le caratteristiche dei file di intestazione della libreria standard.

La linea 3

```
#include <iostream>
```

mostra la nuova sintassi per i nomi dei file di intestazione.

La linea 5

```
using namespace std;
```

indica che stiamo utilizzando lo *spazio dei nomi* `std`: una recente caratteristica del C++. Gli spazi dei nomi sono stati pensati per aiutare i programmatori a sviluppare nuovi componenti software senza generare conflitti con componenti già esistenti. Uno dei problemi che si incontra quando si sviluppano librerie di classi è che i nomi che si scelgono per classi e funzioni possono essere stati già scelti altrove da qualcun altro. Gli spazi dei nomi sono utilizzati per mantenere nomi univoci per ogni nuovo componente software.


```
1  // Fig. 1.16: fig01_16.cpp
2  // Usare i file di intestazione nel nuovo stile
3  #include <iostream>
4
5  using namespace std;
6
7  int main()
8  {
9     cout << "Welcome to C++!\n";
10    std::cout << "Welcome to C++!\n";
11
12    return 0; // indica che il programma e' terminato con successo
13 }
```

```
Welcome to C++!
Welcome to C++!
```

Figura 1.16 Usare i file di intestazione nel nuovo stile.

Ogni file di intestazione, secondo lo standard del C++, utilizza uno spazio dei nomi chiamato `std` per garantire che tutte le caratteristiche della libreria standard non entrino in conflitto con componenti software sviluppati da terze parti. I programmatori non dovrebbero utilizzare `std` per definire nuove librerie di classi. Questa istruzione indica semplicemente che si stanno utilizzando dei componenti software della libreria standard del C++. Se dovessimo definire una nostra libreria di classi, potremmo porre le nostre classi e le nostre funzioni nello spazio dei nomi `deitel`, per renderla univoca rispetto alle librerie di classi di tutti gli altri produttori di software e alla libreria standard del C++.

Una volta inclusa l'istruzione `using namespace std;` possiamo utilizzare direttamente l'oggetto `cout` per inviare i valori sullo stream di output standard. Se si utilizzano due o più librerie di classi che hanno componenti dai nomi identici può crearsi un conflitto di nomi. In questo caso occorre qualificare completamente il nome che si vuole utilizzare, unendolo con lo spazio dei nomi a cui appartiene, come nell'istruzione seguente:

```
std::cout << "Welcome to C++!\n";
```

Il nome di `cout` qualificato completamente è `std::cout`. Se vogliamo utilizzare sempre questa forma (in verità un po' pesante), non abbiamo bisogno di specificare `using namespace std;` nella linea 5 del programma. L'istruzione `using` ci consente di far uso della forma abbreviata per tutti i nomi della libreria standard (o di qualsiasi spazio dei nomi si voglia scegliere). Discuteremo più in dettaglio degli spazi dei nomi in seguito. Non tutti gli ambienti di sviluppo però supportano a tutt'oggi le nuove convenzioni per i file di intestazione. Questo è il motivo che ci ha spinto a utilizzare nel libro il vecchio stile, tranne quando introduciamo le nuove caratteristiche dello standard. Chiariremo comunque in ogni occasione se utilizziamo o meno queste nuove convenzioni.

1.20 Pensare in termini di oggetti: le tecniche orientate agli oggetti e UML (Unified Modeling Language™)

In questa sezione iniziamo il nostro percorso di studio sulle tecniche di progettazione orientate agli oggetti. Come vedremo, esse riflettono il modo naturale di pensare ai problemi reali e di conseguenza ai programmi per computer.

Nei primi cinque capitoli di questo volume analizzeremo la metodologia "convenzionale" della programmazione strutturata, perché gli oggetti che creeremo sono composti da parti costituenti delle porzioni di programma strutturate. Ogni capitolo avrà una sezione conclusiva intitolata "Pensare in termini di oggetti" in cui presenteremo in modo graduale le tecniche di programmazione orientata agli oggetti. Il nostro obiettivo, in queste sezioni, è quello di spingervi ad acquisire una mentalità orientata agli oggetti, per poter sfruttare immediatamente le nozioni di programmazione orientata agli oggetti che esploreremo nel Capitolo 6. Infine introdurremo gradualmente *UML* (*Unified Modeling Language*), il linguaggio standard per la progettazione di sistemi. UML è un linguaggio grafico che consente alle persone coinvolte in un progetto (ingegneri del software, sistemisti, programmatori e così via) di rappresentare i loro progetti utilizzando una notazione comunemente accettata.

In questa sezione introduciamo gli oggetti, illustrandone i concetti fondamentali e la terminologia relativa. Nelle sezioni facoltative "Pensare in termini di oggetti" alla fine dei prossimi quattro capitoli prenderemo in considerazione alcune questioni più sostanziali che riguardano questo argomento, perché cercheremo di risolvere un problema di media complessità con le tecniche di progettazione orientata agli oggetti (OOD, dall'inglese Object-Oriented Design). In sostanza analizzeremo un problema e progetteremo un sistema per risolverlo, determinando gli oggetti necessari per la sua implementazione, insieme con i loro attributi e comportamenti, e specificando inoltre il modo in cui tali oggetti interagiranno reciprocamente perché il sistema funzioni correttamente. Tutto ciò avverrà prima che abbiate imparato a scrivere codice orientato agli oggetti nel linguaggio C++. Successivamente, nelle sezioni "Pensare in termini di oggetti" dei Capitoli 6, 7 e 9 discuteremo l'implementazione in C++ del sistema progettato nei capitoli precedenti.

Lo studio di questo problema vi preparerà ad affrontare i progetti più complessi del mondo reale. Se siete studenti e il vostro docente non ha intenzione di includere queste sezioni opzionali nel vostro corso vi consigliamo di studiarle per conto vostro, perché crediamo che sia un progetto stimolante e di dimensioni non trascurabili per un principiante. Alla fine avrete una buona conoscenza di base della progettazione orientata agli oggetti e del linguaggio UML e avrete una migliore comprensione del codice scritto in C++, perché avrete analizzato punto per punto un programma di certe dimensioni scritto e documentato con attenzione.

Iniziamo dunque a introdurre un po' di terminologia. Adesso guardatevi intorno: dovunque rivolgiate il vostro sguardo li vedete: *oggetti*! Persone, animali, piante, automobili, aerei, palazzi, computer e quant'altro. Gli uomini pensano in termini di oggetti. Abbiamo la capacità di *astrazione*, che ci consente di vedere dei pixel (punti luminosi) su uno schermo come oggetti, persone, animali o cose, piuttosto che come puntini colorati. Possiamo pensare in termini di spiagge anziché di granelli di sabbia, di foreste anziché di alberi e di case anziché di mattoni.

Possiamo suddividere gli oggetti in due grandi categorie: oggetti animati e inanimati. I primi sono vivi, si muovono e intraprendono azioni, al contrario dei secondi: una tovaglia rimane sul tavolo. Tuttavia entrambi i tipi di oggetti hanno una cosa in comune: hanno degli *attributi*, come la dimensione, la forma, il colore e il peso. Inoltre hanno tutti dei *comportamenti*: una palla rotola, rimbalza, si gonfia o si sgonfia; un bambino piange, dorme, gattona, cammina e sbatte le palpebre; un'automobile accelera, frena e svolta; un asciugamano assorbe l'acqua.

Gli esseri umani imparano delle cose sugli oggetti studiando i loro attributi e osservando i comportamenti. Oggetti diversi possono avere attributi e comportamenti molto simili. Per esempio si può fare un confronto tra i bambini e gli adulti, o tra gli esseri umani e gli scimpanzè. Automobili, autocarri e pattini a rotelle hanno molto in comune.

La programmazione a oggetti (OOP, Object Oriented Programming) modella gli oggetti software sulla base degli oggetti del mondo reale. Essa si avvale del concetto di astrazione *classe*, per cui oggetti di una determinata classe (per es. la classe delle automobili) hanno le stesse caratteristiche. Si avvale poi di relazioni di *ereditarietà*, (singola e multipla) secondo le quali nuove classi di oggetti sono derivate da classi esistenti, ereditando le loro caratteristiche e estendendole con caratteristiche proprie. Ad esempio, un oggetto della classe automobile decappottabile ha le stesse caratteristiche di un oggetto della classe automobile, ma il suo tetto si può aprire e chiudere.

La programmazione a oggetti ci dà un modo più naturale e intuitivo di pensare al processo della programmazione, *modellando* gli oggetti del mondo reale, i loro attributi e i loro comportamenti. La OOP modella anche la comunicazione tra gli oggetti. Essi comunicano tra loro tramite messaggi, proprio come le persone (per es. come un sergente che mette la truppa sull'Attenti!).

La OOP *incapsula* i dati (gli attributi) e le funzioni (i comportamenti) in pacchetti detti *oggetti*; i dati e le funzioni di un oggetto sono intimamente correlati. Gli oggetti hanno anche la proprietà di *tenere nascoste le informazioni*. Ciò significa che sebbene gli oggetti possano sapere come comunicare tra loro attraverso *interfacce* ben definite, non sempre hanno la possibilità di conoscere la struttura interna degli altri oggetti: i dettagli di implementazione sono nascosti all'interno di ciascun oggetto. È possibile, infatti, guidare un'automobile senza sapere come funziona il motore, la trasmissione e i dettagli degli altri sistemi interni. Vedremo in seguito perché nascondere le informazioni è di importanza cruciale per la creazione di software di buona qualità.

Nel C e negli altri *linguaggi procedurali*, la programmazione tende a essere *orientata all'azione*, mentre in C++ tende a essere *orientata agli oggetti*. In C l'unità di programmazione è la *funzione*. In C++ l'unità è la *classe*, da cui eventualmente si *istanziano* (creano) oggetti. Le classi del C++ contengono funzioni dette *metodi*.

I programmatori C si concentrano sulla scrittura di funzioni, cioè di gruppi di azioni che effettuano un'operazione completa. Un programma è costituito da un insieme di funzioni. I dati sono importanti anche in C, naturalmente, ma il concetto è che essi siano

soltanto un supporto alle azioni da intraprendere. I *verbi* presenti nelle specifiche (descrizioni) di un sistema guidano il programmatore C nel determinare l'insieme di funzioni che devono lavorare insieme per implementarlo.

Chi programma in C++, invece, concentra l'attenzione sulla creazione di nuovi tipi di dato, detti *classi*. Una classe contiene sia i dati che le funzioni deputate a manipolarli. I componenti di una classe che sono dati sono detti *dati membro*; allo stesso modo le funzioni sono dette *funzioni membro* o *metodi*. L'istanza di un tipo di dato predefinito come `int` è detta *variabile*: allo stesso modo l'istanza di un tipo di dato definito dall'utente, come una classe, è detto *oggetto*. Il programmatore utilizza i tipi di dato predefiniti come mattoni per la costruzione dei propri tipi di dato. Con questo approccio i programmatori C++ sono guidati nel determinare l'insieme delle classi necessarie dai *nomi* presenti nelle specifiche di un sistema , dalle classi si creano poi gli oggetti che opereranno in sinergia per implementare il sistema. La relazione fra classi ed oggetti può essere riassunta nel modo seguente: i progetti edilizi stanno alle case allo stesso modo in cui le classi stanno agli oggetti. Infatti, è possibile costruire parecchie case partendo dallo stesso progetto e, allo stesso modo, possiamo istanziare parecchi oggetti da una sola classe.Le classi possono inoltre presentare relazioni con altre classi: per esempio, nella concettualizzazione orientata agli oggetti di una banca, la classe `ImpiegatoAlloSportello` sarebbe necessariamente essere in relazione con la classe `Cliente`. Tali relazioni prendono il nome di *associazioni*.

Le classi che compongono un software possono essere riutilizzate in un secondo momento in altri sistemi. Sono in circolazione dei pacchetti di componenti riutilizzabili che contengono per l'appunto gruppi di classi correlate. Se ci consentite un paragone, così come gli agenti immobiliari enfatizzano la parola "locazione" per l'influenza che ha sul prezzo dei fabbricati, in modo analogo noi enfatizziamo una sola parola per l'influenza che avrà sul futuro del software: "riutilizzo"

Grazie alla tecnologia orientata agli oggetti potremo scrivere la maggior parte del software futuro semplicemente combinando parti standardizzate e intercambiabili, le classi per l'appunto. Lo scopo di questo corso è insegnarvi a creare classi utili e riutilizzabili. Ogni volta che si crea una nuova classe, essa può diventare potenzialmente un piccolo patrimonio software, che può essere riutilizzato anche da terzi per velocizzare o migliorare la qualità del proprio software. Ed è indubbio che questa sia una gran bella potenzialità.

1.20.1 Introduzione all'analisi e alla progettazione orientate agli oggetti (OOAD)

A questo punto del corso probabilmente avrete già scritto alcuni programmi in C++ di modeste dimensioni. Provate ora a pensare come avete creato il codice dei vostri programmi. Molto probabilmente avrete acceso il computer e avrete semplicemente cominciato a digitare delle istruzioni in sequenza. Questo approccio, però, funziona solamente per piccoli progetti: che cosa fareste se vi chiedessero di creare il sistema software che controlla gli sportelli bancomat di una grossa banca? Un progetto del genere è troppo esteso e complicato perché si possa affrontarlo sedendosi al computer e digitando un'istruzione dopo l'altra.

Se si vogliono soluzioni efficaci occorre seguire un qualche procedimento che stabilisca in modo chiaro quali sono i requisiti del sistema, per poter stendere di conseguenza un progetto che li soddisfi nel modo migliore. Nel caso di un programma complesso, quindi, prima di scrivere una sola istruzione di codice dovrete seguire interamente tale procedimento e chiedere preventivamente l'approvazione del progetto da parte dei vostri superiori. Tale procedimento, se svolto nell'ottica orientata agli oggetti viene chiamato *analisi e progettazione orientata agli oggetti* (*OOAD*, per Object-Oriented Analysis and Design).

Gli sviluppatori esperti sanno che non ha importanza quanto semplice possa sembrare un progetto ma che è meglio impiegare un po' di tempo nell'analisi e nella progettazione. Questo tempo è, in un certo senso, risparmiato, perché capita spesso che, non effettuando un'accurata analisi e progettazione, si abbandoni un approccio che si rivela inadatto per abbracciarne un altro a metà dell'implementazione.

"OOAD" è un termine generico che designa le idee guida di analisi e progettazione del sistema per la soluzione di un dato problema. I piccoli problemi come quelli che presentiamo nei primi capitoli del nostro corso non richiedono, in realtà, un procedimento troppo approfondito e lo scrivere lo pseudocodice prima del codice vero e proprio sarà più che sufficiente (Nota: lo pseudocodice è un mezzo che riflette in modo informale il codice di un programma: non è un vero linguaggio di programmazione, ma serve come falsariga per scrivere successivamente il codice; introdurremo questo concetto nel Capitolo 2). Questo approccio è senz'altro adatto a problemi di piccole dimensioni, ma col crescere delle dimensioni e dei gruppi di lavoro conviene utilizzare i metodi della OOAD. Idealmente un gruppo di lavoro deve trovare un accordo su un procedimento di analisi definito formalmente e su un modo uniforme per comunicarne i risultati . Esistono diversi procedimenti OOAD, ma c'è un linguaggio grafico per comunicarne i risultati che sta conoscendo un largo consenso: è *UML*, acronimo inglese di *Unified Modeling Language*, linguaggio unificato per la modellazione di sistemi. UML è stato sviluppato verso la metà degli anni '90, sotto la guida iniziale di tre studiosi di ingegneria del software: Grady Booch, James Rumbaugh and Ivar Jacobson.

1.20.2 Storia di UML

Negli anni '80 sempre più organizzazioni cominciarono a utilizzare la OOP per le loro applicazioni, e di conseguenza si presentò la necessità di decidere un procedimento condiviso per la OOAD. Diversi studiosi di ingegneria del software, tra cui Booch, Rumbaugh e Jacobson, proposero individualmente dei procedimenti distinti, ognuno dei quali prevedeva una notazione propria, o "linguaggio" in forma di diagrammi grafici, che serviva a presentare i risultati dell'analisi e della progettazione.

Nei primi anni '90 aziende diverse, e persino settori diversi di una stessa azienda, utilizzavano notazioni e procedimenti diversi. Per complicare le cose, tali aziende richiedevano strumenti software che supportassero i loro procedimenti specifici, ma data la diversità di procedimenti diversi, i produttori di software trovavano alquanto difficile soddisfare le loro richieste. Ovviamente c'era bisogno di stabilire uno standard.

Nel 1994 James Rumbaugh raggiunse Grady Booch presso la "Rational Software Corporation", e i due cominciarono a lavorare all'unificazione dei procedimenti di analisi e sviluppo che utilizzavano comunemente. A loro si unì poco dopo anche Ivar Jacobson. Nel 1996 questo gruppo definì la prima versione di UML, rivolto alla comunità di programmatori, a cui il gruppo chiedeva anche un feedback. Più o meno nello stesso periodo l'organizzazione *Object Management Group*™ (*OMG*™) invitò la comunità a proporre un

linguaggio comune di rappresentazione dei sistemi. L'OMG è un'organizzazione no-profit che promuove l'utilizzo delle tecniche orientate agli oggetti, elaborando guide e specifiche. Diversi gruppi industriali (tra cui HP, IBM, Microsoft, Oracle e Rational Software) avevano già compreso l'importanza di un linguaggio comune per la rappresentazione dei sistemi, e in risposta alla richiesta di standard avanzata dall'OMG, essi si unirono formando il consorzio degli *UML Partners*, proponendo la versione 1.1 di UML. L'OMG giudicò positivamente la proposta e nel 1997 si assunse la responsabilità di definire e rivedere le versioni e future di UML. Nel 1999 l'OMG ha definito la versione 1.3 di UML, che è la versione corrente nel momento in cui questo testo è stato redatto.

1.20.3 Che cos'è UML?

Unified Modeling Language è ora uno degli schemi di rappresentazione grafica più utilizzati per rappresentare i sistemi orientati agli oggetti: esso ha infatti unificato diverse notazioni che esistevano già alla fine degli anni '80. I progettisti si servono di questo linguaggio per rappresentare graficamente i sistemi a cui lavorano.

Una delle caratteristiche più interessanti di UML è la sua flessibilità. UML è estendibile ed è indipendente dai procedimenti di analisi e progettazione utilizzati. Chi utilizza UML è libero di sviluppare i sistemi servendosi di diversi procedimenti, ciò che conta è che può ora esprimere le specifiche di tali sistemi tramite un insieme standard di notazioni.

UML è un linguaggio grafico ricco e complesso. Nelle sezioni "Pensare in termini di oggetti" ne presenteremo una versione ridotta e semplificata, e la utilizzeremo per introdurvi all'argomento della progettazione orientata agli oggetti. Per una discussione più completa su UML potete fare riferimento al sito dell'OMG (http://www.omg.org) e al documento che contiene le specifiche della versione 1.3 (http://www.amp.org/uml/).

Sono stati pubblicati diversi libri su UML: il testo di Martin Fowler e Kendall Scott "UML Distilled", seconda edizione, introduce la versione 1.3 di UML in dettaglio, fornendo anche diversi esempi. Il testo di Booch, Rumbaugh e Jacobson "Unified Modeling Language User Guide" è invece *la* guida di riferimento ad UML.

Le tecniche orientate agli oggetti sono molto diffuse oggi nell'industria del software, e la stessa sorte sta toccando ora ad UML. Il nostro obiettivo nelle sezioni "Pensare in termini di oggetti" è insegnarvi a pensare in termini di oggetti il più velocemente possibile. Dalla sezione "Pensare in termini di oggetti" del Capitolo 2 comincerete ad applicare queste tecniche ad un problema di dimensioni non trascurabili. E speriamo realmente che possiate trovare il progetto che proponiamo divertente e stimolante.

Esercizi di autovalutazione

1.1 Completate le seguenti frasi:
a) L'azienda che ha reso popolari i personal computer è stata la _____.
b) Fra i primi computer introdotti nelle aziende e nelle industrie troviamo il _____
c) I computer elaborano i dati sotto il controllo di insiemi di istruzioni detti _____
d) Le sei unità logiche principali di un computer sono _____, _____, _____, _____, _____ e _____
e) Le tre classi di linguaggi che abbiamo introdotto in questo capitolo sono _____, _____, e _____

f) I programmi che convertono i programmi scritti in linguaggi ad alto livello in linguaggio macchina si chiamano _____
g) C è noto principalmente come il linguaggio in cui è stato scritto il sistema operativo
h) Il linguaggio _____ è stato sviluppato da Wirth per l'insegnamento della programmazione strutturata nelle Università.
i) Il Dipartimento della difesa degli Stati Uniti sviluppò il linguaggio Ada, con una funzionalità detta _____ che consente ai programmatori di specificare più di un'attività da eseguire in parallelo.

1.2 Completate le seguenti affermazioni che riguardano l'ambiente di sviluppo C++.
a) I programmi in C++ sono normalmente digitati alla tastiera utilizzando un programma detto _____
b) In un ambiente di sviluppo C++, prima della compilazione viene eseguito un programma detto _____
c) Il _____ combina l'output prodotto dal compilatore con le varie funzioni di libreria, per produrre un'immagine eseguibile.
d) Il _____ trasferisce l'immagine eseguibile di un programma in C++ dal disco alla memoria.

1.3 Completate le seguenti frasi:
a) Ogni programma in C++ comincia dalla funzione _____
b) Il corpo di ogni funzione inizia con _____ e termina con _____
c) Ogni istruzione termina con un _____
d) La sequenza di escape \n rappresenta il carattere _____, che sposta il cursore sulla riga successiva dello schermo.
e) L'istruzione _____ serve a prendere delle decisioni.

1.4 Stabilite quali delle seguenti affermazioni sono vere. Per le altre, spiegate perché sono false.
a) Se un programma contiene dei commenti, durante l'esecuzione verrà visualizzato il testo che segue il segno `//`.
b) La sequenza di escape \n inviata in output con `cout`, fa spostare il cursore all'inizio della riga successiva dello schermo.
c) Tutte le variabili devono essere dichiarate prima di poter essere utilizzate.
d) Ogni variabile deve appartenere a un tipo di dato determinato, e questo deve essere specificato durante la sua dichiarazione.
e) C++ considera gli identificatori `number` e `NuMbEr` come la stessa variabile.
f) Le dichiarazioni possono comparire quasi ovunque all'interno di una funzione.
g) L'operatore modulo (`%`) può essere utilizzato solo con operandi interi.
h) Gli operatori aritmetici `*`, `/`, `%`, `+` e `-` hanno tutti lo stesso livello di precedenza.
i) Un programma in C++ per visualizzare tre linee di output deve contenere tre istruzioni che utilizzano `cout`.

1.5 Scrivete una istruzione in C++ che fa ciò che segue:
a) Dichiara le variabili `c`, `thisIsAVariable`, `q76354` e `number`, tutte di tipo `int`.
b) Chiede all'utente di immettere un numero intero. Il messaggio visualizzato dovrebbe terminare con il segno di due punti (`:`) e uno spazio, e il cursore dovrebbe restare nella posizione subito dopo lo spazio.
c) Legge un intero dalla tastiera e memorizza il valore nella variabile intera `age`.
d) Se la variabile `number` non è uguale a 7, visualizza `"La variabile number non è uguale a 7"`
e) Visualizza su una linea il messaggio `"Questo programma è stato scritto in C++"`

f) Visualizza su due linee il messaggio `"Questo programma è stato scritto in C++"`. La prima linea termina dopo la parola `programma`.
g) Visualizza il messaggio `"Questo programma è stato scritto in C++"`, ponendo ogni parola su una linea diversa.
h) Visualizza il messaggio `"Questo programma è stato scritto in C++"` separando ogni parola con un carattere di tabulazione.

1.6 Scrivete un'istruzione (o un commento) che fa ciò che segue:
a) Informa l'utente che il programma calcola il prodotto di tre numeri interi.
b) Dichiara le variabili `x`, `y`, `z` e `result` di tipo `int`.
c) Chiede all'utente di digitare tre numeri interi.
d) Legge tre interi dalla tastiera e li memorizza nelle variabili `x`, `y` e `z`.
e) Calcola il prodotto dei tre interi contenuti nella variabili `x`, `y` e `z`, e assegna il risultato alla variabile `result`.
f) Visualizza `"Il prodotto è "` e di seguito il valore della variabile `result`.
g) Restituisce un valore dall'interno della funzione `main` che indica che il programma è stato eseguito con successo.

1.7 Utilizzate le istruzioni che avete scritto nell'Esercizio 1.6 per scrivere un programma completo e funzionante che calcoli il prodotto di tre numeri interi.

1.8 Identificate e correggete gli errori presenti nelle seguenti istruzioni:

```
a) if ( c < 7 );
       cout << "c è minore di 7\n";
b) if ( c => 7 )
       cout << "c è maggiore o uguale a 7\n";
```

1.9 Completate le affermazioni seguenti con il termine più appropriato:
a) Gli esseri umani sono in grado di vedere i puntini luminosi di uno schermo televisivo, oppure possono rielaborare l'immagine che percepiscono, vedendo tre persone sedute a un tavolo. Questa capacità si chiama _____
b) Se vediamo un'automobile come un oggetto, il fatto se è decappottabile è un attributo/comportamento (scegliete quella giusta) _____ dell'automobile.
c) Il fatto che un'automobile possa accelerare o decelerare, girare a sinistra o a destra oppure andare avanti o indietro sono tutti esempi di _____ dell'oggetto automobile.
d) Quando un nuovo tipo di classe eredita delle caratteristiche da molte classi diverse che già esistono, chiamiamo questo tipo di ereditarietà _____
e) Gli oggetti comunicano l'uno con l'altro inviandosi _____
f) Gli oggetti comunicano l'uno con l'altro su _____ ben definiti.
g) Ogni oggetto, normalmente, non conosce i dettagli di implementazione degli altri oggetti. Questa proprietà è detta _____
h) Il _____ nella specifica di un sistema è d'aiuto al programmatore per determinare le classi di cui avrà bisogno per implementare il sistema in questione.
i) I dati di una classe sono detti _____ e le funzioni sono dette _____
j) Un'istanza di un tipo definito dall'utente è detta _____

Risposte agli esercizi di autovalutazione

1.1 a) Apple. b) IBM Personal Computer. c) programmi. d) unità di input, unità di output, memoria, unità aritmetico-logica, unità centrale di elaborazione, unità di memorizzazione secondaria. e) linguaggi macchina, linguaggi assembly, linguaggi ad alto livello. f) compilatori. g) UNIX. h) Pascal. i) multitasking.

1.2 a) editor. b) preprocessore. c) linker. d) loader.

1.3 a) `main`. b) parentesi graffa aperta ({), parentesi graffa chiusa (}). c) punto e virgola. d) carattere di nuova linea. e) `if`.

1.4 a) Falsa. I commenti sono semplicemente ignorati dal compilatore. Essi sono utili a documentare i programmi e renderli più comprensibili.
b) Vera.
c) Vera.
d) Vera.
e) Falsa. Il C++ fa differenza tra lettere maiuscole e minuscole, per cui i due identificatori sono due variabili diverse.
f) Vera.
g) Vera.
h) Falsa. Gli operatori `*`, `/` e `%` hanno la stessa precedenza mentre gli operatori `+` e `-` hanno precedenza più bassa.
i) Falsa. Una sola istruzione di output che fa uso di `cout` può visualizzare più linee di testo, basta utilizzare più sequenze di escape `\n`.

1.5 a)
```
int c, thisIsAVariable, q76354, number;
```
b)
```
cout << "Enter an integer: ";
```
c)
```
cin >> age;
```
d)
```
if ( number != 7 )
   cout << "La variabile number non è uguale a 7\n";
```
e)
```
cout << "Questo programma è stato scritto in C++\n";
```
f)
```
cout << "Questo programma\né stato scritto in C++ \n";
```
g)
```
cout << "Questo\nprogramma\nè\nstato\nscritto\nin\nC++\n";
```
h)
```
cout << " Questo\tprogramma\tè\tstato\tscritto\tin\tC++\n";
```

1.6 a)
```
// Calcola il prodotto di tre interi
```
b)
```
int x, y, z, result;
```
c)
```
cout << "Immetti tre numeri interi: ";
```
d)
```
cin >> x >> y >> z;
```
e)
```
result = x * y * z;
```
f)
```
cout << "Il prodotto è " << result << endl;
```
g)
```
return 0;
```

1.7
```
// Calcola il prodotto di tre interi
#include <iostream.h>

int main()
{
   int x, y, z, result;

   cout << "Immetti tre interi: ";
   cin >> x >> y >> z;
   result = x * y * z;
   cout << "Il prodotto e " << result << endl;

   return 0;
}
```

1.8 a) Errore: il punto e virgola dopo la parentesi tonda chiusa, nella condizione della `if`. Correzione: Eliminare il punto e virgola. Nota: Il risultato di questo errore è che l'istruzione di output viene eseguita in ogni caso, anche se la condizione della `if` è falsa. Il punto e virgola, infatti, viene considerato come un'istruzione vuota, cioè un'istruzione *che non esegue* alcun compito. Parleremo di nuovo dell'istruzione vuota nel prossimo capitolo.
b) Errore: L'operatore relazionale => Correzione: Al posto di => scrivere >=

1.9 a) astrazione. b) attributo. c) comportamenti. d) multipla. e) messaggi. f) interfacce. g) occultamento delle informazioni. h) nomi. i) dati membro; funzioni membro o metodi. j) oggetto.

Esercizi

1.10 Per ogni voce indicate se si tratta di hardware o di software:
a) CPU
b) Compilatore C++
c) Unità aritmetico-logica (ALU)
d) Preprocessore C++
e) Unità di input
f) Programma editor

1.11 Per quali ragioni si preferisce scrivere un programma in un linguaggio indipendente dalla macchina, anziché in uno dipendente? Per quali ragioni un linguaggio dipendente dalla macchina può rivelarsi più appropriato per alcuni tipi di programmi?

1.12 Completate le seguenti affermazioni:
a) Quale unità logica di un computer riceve i dati dall'esterno? _____
b) Il procedimento con cui si indica al computer di risolvere un problema specifico è detto _____
c) Che tipo di linguaggio di programmazione utilizza delle abbreviazioni di termini inglesi come istruzioni? _____
d) Qual è l'unità logica del computer che invia i dati che sono stati elaborati alle varie periferiche, in modo che possano essere utilizzati all'esterno del computer? _____
e) Qual è l'unità logica del computer che conserva i dati? _____
f) Qual è l'unità logica del computer che effettua i calcoli? _____
g) Qual è l'unità logica del computer deputata ai processi logico/decisionali? _____
h) Il livello di linguaggio più conveniente per la scrittura delle applicazioni, in termini di velocità e di semplicità, è _____
i) L'unico linguaggio che un computer comprende direttamente è il suo _____
j) Qual è l'unità logica del computer che coordina le attività di tutte le altre unità? _____

1.13 Spiegate a che servono i seguenti oggetti:
a) `cin`
b) `cout`
c) `cerr`

1.14 Perché la programmazione orientata agli oggetti è così importante al giorno d'oggi?

1.15 Completate le seguenti affermazioni:
a) _____ servono a documentare i programmi e a renderli più leggibili.
b) L'oggetto che si utilizza per visualizzare i dati sullo schermo è _____
c) Un'istruzione C++ per prendere una decisione basata su una condizione logica è _____
d) Normalmente i calcoli sono effettuati dalle istruzioni _____
e) L'oggetto _____ riceve dei valori in input dalla tastiera.

1.16 Scrivete una sola istruzione in C++, o una sola linea, che fa ciò che segue:
a) Visualizza il messaggio `"Immetti due numeri"`
b) Assegna il prodotto delle variabili `b` e `c` alla variabile `a`.
c) Informa l'utente che il programma effettua dei calcoli finanziari (utilizzate, per esempio, del testo che documenti un programma).
d) Riceve tre valori in input dalla tastiera e li memorizza nelle variabili intere `a`, `b` e `c`.

1.17 Stabilite quali affermazioni sono vere e quali sono false. Commentate brevemente le vostre risposte.
a) Gli operatori del C++ associano da sinistra verso destra.
b) Questi sono tutti nomi di variabili validi: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
c) L'istruzione `cout << "a = 5;";` è un tipico esempio di istruzione di assegnamento.
d) Un'espressione aritmetica valida senza parentesi viene normalmente calcolata da sinistra a destra.
e) I seguenti nomi di variabile non sono validi: `3g`, `87`, `67h2`, `h22`, `2h`.

1.18 Completate le seguenti affermazioni:
a) Quali operazioni aritmetiche hanno lo stesso livello di precedenza della moltiplicazione?
_____
b) Nel caso di parentesi nidificate, quale coppia di parentesi viene calcolata per prima in un'espressione aritmetica? _____
c) Una locazione di memoria che può contenere valori diversi in momenti diversi, durante l'esecuzione di un programma, si chiama _____

1.19 Che cosa visualizzano le seguenti istruzioni in C++? Se un'istruzione non visualizza nulla *rispondete "nulla". Per ipotesi* `x = 2` *e* `y = 3`.

```
a) cout << x;
b) cout << x + x;
c) cout << "x=";
d) cout << "x = " << x;
e) cout << x + y << " = " << y + x;
f) z = x + y;
g) cin >> x >> y;
h) // cout << "x + y = " << x + y;
h) cout << "\n";
```

1.20 Quali delle seguenti istruzioni modifica il valore delle proprie variabili?
a) `cin >> b >> c >> d >> e >> f;`
b) `p = i + j + k + 7;`
c) `cout << "variabili i cui valori sono stati distrutti";`
d) `cout << "a = 5";`

1.21 Data l'equazione algebrica $y = ax^3 + 7$, quale delle seguenti istruzioni la rappresenta correttamente in C++? (Non garantiamo che ce ne sia almeno una)

```
a) y = a * x * x * x + 7;
b) y = a * x * x * (x + 7);
c) y = (a * x) * x * (x + 7);
d) y = (a * x) * x * x + 7;
e) y = a * (x * x * x) + 7;
f) y = a * x * (x * x + 7);
```

1.22 Stabilite l'ordine di calcolo degli operatori in queste istruzioni, e visualizzate il valore di x dopo l'esecuzione di ognuna di esse.

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = (3 * 9 * (3 + (9 * 3/ (3))));`

1.23 Scrivete un programma che chiede all'utente di immettere due numeri, legge i due numeri e ne visualizza la somma, il prodotto, la differenza e il quoziente.

1.24 Scrivete un programma che visualizza i numeri da 1 a 4 sulla stessa linea, con ogni coppia di numeri adiacenti separata da uno spazio. Scrivetelo con uno dei metodi seguenti:

a) Utilizzando un'istruzione di output e un solo operatore di inserzione nello stream.
b) Utilizzando un'istruzione di output e quattro operatori di inserzione nello stream.
c) Utilizzando quattro istruzioni di output.

1.25 Scrivete un programma che chiede all'utente di digitare due numeri interi, legge i due numeri e visualizza il più grande seguito dalle parole `"è il più grande"`. Se i numeri sono uguali, visualizzate il messaggio `"I numeri sono uguali"`

1.26 Scrivete un programma che chiede all'utente di digitare tre interi, e ne visualizza la somma, la *media, il prodotto, il numero più piccolo* e *il* più grande. Ecco che aspetto dovrebbe avere l'output del programma:

```
Immetti tre interi diversi: 13 27 14
La somma è 54
La media e 18
Il prodotto è 4914
Il più piccolo è 13
Il più grande e 27
```

1.27 Scrivete un programma che legge il raggio di un cerchio e visualizza il diametro, la circonferenza e l'area del cerchio. Per pi greco utilizzate il valore costante 3.14159. Effettuate i calcoli nelle istruzioni di output. Nota: in questo capitolo abbiamo parlato soltanto delle costanti e delle variabili intere. Nel Capitolo 3 parleremo dei numeri a virgola mobile, cioè dei valori che possono avere una parte decimale.

1.28 Scrivete un programma che visualizza un rettangolo, un'ellisse, una freccia e un rombo di questo tipo:

1.29 Che cosa visualizza la seguente istruzione?

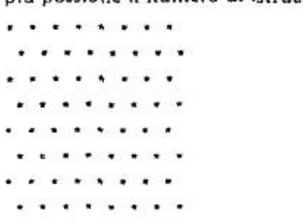
```
cout << "**\n**\n***\n****\n*****\n";
```

1.30 Scrivete un programma che legge cinque interi e ne determina il valore minore e quello maggiore. Utilizzate solo le tecniche che avete imparato in questo capitolo.

1.31 Scrivete un programma che legge un intero e determina se è pari o dispari. Suggerimento: usate l'operatore modulo. Un numero pari è sempre multiplo di due. I multipli di due danno resto zero quando *sono divisi per 2*.

1.32 Scrivete un programma che legge due interi e determina se il primo è multiplo del secondo. Suggerimento: usate l'operatore modulo.

1.33 Visualizzate una scacchiera con otto istruzioni di output. Successivamente cercate di ridurre il più possibile il numero di istruzioni.



1.34 *Che differenza c'è tra errore fatale e non fatale? In che senso è meglio incorrere in un errore* fatale, piuttosto che in uno non fatale?

1.35 Facciamo un passettino avanti. In questo capitolo abbiamo parlato degli interi e del tipo `int`. In C++ è possibile rappresentare anche le lettere minuscole e maiuscole, assieme a tanti altri simboli speciali. Il C++ utilizza dei piccoli numeri interi per rappresentare i caratteri. È il set di caratteri del computer su cui siete a decidere la corrispondenza tra i caratteri e questi valori numerici. Potete visualizzare un carattere scrivendolo semplicemente tra apici singoli, in questo modo

```
cout << 'A',
```

Potete visualizzare il valore intero corrispondente a un dato carattere precedendolo con `(int)`: questa operazione è detta *cast* (ne parleremo nel Capitolo 2):

```
cout << (int) 'A';
```

Quando il computer esegue questa istruzione, visualizza il valore 65 (sui sistemi che utilizzano il cosiddetto set di caratteri ASCII). Scrivete un programma che visualizza il valore intero che corrisponde alle lettere maiuscole, minuscole, alle cifre numeriche e ad alcuni simboli speciali. Come minimo determinate il valore corrispondente a `A B C a b c 0 1 2 $ * + /` e allo spazio.

1.36 Scrivete un programma che riceve in input un numero di cinque cifre, separa il numero nelle cifre che lo compongono e le visualizza distanziandone ognuna di tre spazi dall'altra. Suggerimento: usate la divisione intera e l'operatore modulo. Per esempio, se l'utente digita `42339` il programma visualizzerà

```
4   2   3   3   9
```

1.37 Utilizzando solo le tecniche che avete imparato in questo capitolo, scrivete un programma che calcola il quadrato e il cubo dei numeri da 0 a 10 e li visualizza in una tabella, come segue:

```
numero  quadrato  cubo
0       0         0
1       1         1
2       4         8
3       9         27
4       16        64
5       25        125
6       36        216
7       49        343
8       64        512
9       81        729
10      100       1000
```

1.38 Rispondete brevemente a queste domande sul concetto di "oggetto"

a) Perché questo testo sceglie di parlare in dettaglio della programmazione strutturata prima di passare a trattare in modo approfondito la programmazione orientata agli oggetti?

b) Quali sono i passi tipici della progettazione orientata agli oggetti?
c) Come si presenta l'ereditarietà multipla nel mondo reale degli esseri umani?
d) Che genere di messaggi si inviano le persone quando comunicano tra di loro?
e) Gli oggetti si inviano messaggi secondo delle interfacce ben definite. Che interfacce presenta un'autoradio (primo oggetto) al suo proprietario (secondo oggetto)?

1.39 Probabilmente avrete al vostro polso uno degli oggetti più comuni al mondo, un orologio. Discutete come i termini e i concetti tipici degli oggetti software si applicano al vostro orologio: oggetto, attributi, comportamenti, classe, ereditarietà (considerate, per esempio, una radiosveglia), astrazione, modello, messaggi, incapsulamento, interfaccia, occultamento delle informazioni, dati membro e funzioni membro.

CAPITOLO 2

Le strutture di controllo

Obiettivi

- Comprendere le tecniche fondamentali per risolvere i problemi
- Imparare a sviluppare gli algoritmi attraverso passi di raffinamento top-down
- Conoscere i costrutti decisionali if, if/else e switch, che consentono di scegliere tra diverse azioni alternative
- Conoscere i costrutti iterativi while, do/while, che consentono di eseguire ripetutamente un insieme di istruzioni
- Comprendere le iterazioni controllate da contatori e da valori sentinella
- Imparare a utilizzare gli operatori logici, di incremento, di decremento e di assegnamento
- Imparare a utilizzare le istruzioni di controllo break e continue

2.1 Introduzione

Per risolvere un problema di qualsiasi genere è essenziale averne una comprensione adeguata e affrontarlo con una certa consapevolezza e organizzazione. Se volete scrivere un programma che lo risolva, avrete poi bisogno di comprendere il tipo di blocchi da utilizzare e di conoscere alcuni principi di programmazione. In questo secondo capitolo parleremo proprio di questo argomento, e vi presenteremo i principi fondamentali della programmazione strutturata. Le tecniche che illustriamo qui si possono applicare a gran parte dei linguaggi ad alto livello, non soltanto al C++. Nel Capitolo 6, che introduce la programmazione orientata agli oggetti, vedremo come utilizzare le strutture di controllo per creare e manipolare gli oggetti.

2.2 Gli algoritmi

Qualsiasi problema può essere risolto intraprendendo una serie di azioni in un ordine specifico. Una *procedura* che risolve un dato problema in termini di

1. *azioni*
2. *ordine* di esecuzione delle azioni

prende il nome di *algoritmo*. L'esempio che segue mostra quanto sia importante l'ordine delle azioni.

Consideriamo l'algoritmo "al mattino" di un giovane uomo d'affari che deve alzarsi dal letto e andare al lavoro: (1) alzarsi dal letto, (2) togliersi il pigiama, (3) farsi la doccia, (4) vestirsi, (5) fare colazione, (6) dirigersi al lavoro. Questa routine mette in condizioni l'uomo d'affari di essere pronto a prendere decisioni critiche. Supponiamo che una mattina l'ordine delle azioni risulti, come per incanto, modificato leggermente: (1) alzarsi dal letto, (2) togliersi il pigiama, (3) vestirsi, (4) farsi la doccia, (5) fare colazione, (6) dirigersi al lavoro.

Come vedete, basta così poco per spedire un povero uomo d'affari al lavoro in una tenuta umidiccia. L'ordine di esecuzione delle azioni specificate in un programma si chiama *controllo del programma*. In questo capitolo vedremo come agire sul controllo del programma in C++.

2.3 Lo pseudocodice

Lo *pseudocodice* è un linguaggio artificiale e informale che i programmatori adottano per mettere su carta gli algoritmi che hanno in mente. Lo pseudocodice che presentiamo qui è utile in particolar modo per creare algoritmi che si possano convertire facilmente in programmi C++ strutturati. Lo pseudocodice altro non è che una forma di linguaggio naturale: è semplice e amichevole, e non è un vero linguaggio di programmazione.

I programmi scritti in pseudocodice, infatti, non sono destinati ai computer, ma agli uomini, e sono uno strumento valido per mettere nero su bianco le idee fondamentali di un algoritmo. Gli esempi di questo capitolo vi aiuteranno a utilizzare lo pseudocodice in modo efficace.

Abbiamo scelto di presentarvi uno stile di pseudocodice composto unicamente di caratteri, così potrete digitarlo con qualsiasi editor di testo. Se scrivete l'idea di un programma prima con lo pseudocodice, sarà semplice convertirlo poi in C++. In molti casi l'unica cosa da fare è sostituire le istruzioni in pseudocodice con le loro equivalenti del C++.

In un programma in pseudocodice vanno incluse soltanto le istruzioni eseguibili: le *dichiarazioni*, per esempio, non essendo istruzioni eseguibili, vanno tralasciate. Per esempio, l'unico compito dell'istruzione

```
int i;
```

è di informare il compilatore che la variabile `i` è di tipo `int`, e che quindi ha bisogno di un determinato spazio in memoria: in fase di esecuzione non effettua alcuna operazione, come può essere un input, un output o un calcolo matematico. Ad ogni modo, è una nostra scelta quella di adottare questa regola nella scrittura dello pseudocodice, ma ci sono alcuni programmatori che non tralasciano né variabili né commenti.

2.4 Le strutture di controllo

Normalmente le istruzioni di un programma vengono eseguite una dopo l'altra, nell'ordine in cui sono scritte. Si parla di *esecuzione sequenziale*. Ci sono però diverse istruzioni del C++ che consentono ai programmatori di variare quest'ordine predefinito, indicando che l'esecuzione deve saltare a un'istruzione diversa da quella scritta subito dopo quella corrente. In questo caso si parla di *trasferimento del controllo*.

Nel corso degli anni '60 il mondo dell'informatica si rese conto che l'uso indiscriminato dei trasferimenti di controllo era la causa dei guai più comuni dei gruppi di lavoro di programmazione. L'istruzione alla sbarra era la famigerata `goto`, che può trasferire il controllo a una qualsiasi altra istruzione del programma, indipendentemente da dove quest'ultima sia localizzata. Implicitamente *programmazione strutturata* divenne sinonimo di *eliminazione del* `goto`.

Böhm e Jacopini dimostrarono che qualsiasi programma può essere riscritto in termini di tre sole *strutture di controllo*, ovvero in termini di *struttura sequenziale*, *struttura di selezione* e *struttura di iterazione*. La struttura sequenziale è implicita in C++. In mancanza di indicazioni diverse il computer esegue le istruzioni una dopo l'altra, nell'ordine in cui le legge nel codice. Il segmento di *diagramma di flusso* in Figura 2.1 mostra una tipica struttura sequenziale che effettua due calcoli nell'ordine dato.

Figura 2.1 Diagramma di flusso della struttura sequenziale.

Un diagramma di flusso rappresenta graficamente un algoritmo o parte di esso. I diagrammi di flusso contengono simboli dal significato preciso, come rettangoli, ovali, cerchietti e rombi: questi simboli sono connessi tra loro tramite linee, dette *linee di flusso*.

Anche i diagrammi di flusso, come lo pseudocodice, servono a rappresentare e a sviluppare gli algoritmi, anche se la maggior parte dei programmatori ha una spiccata preferenza per lo pseudocodice. I diagrammi di flusso hanno la qualità di chiarire anche a livello visivo come operano le strutture di controllo, ed è per questo motivo che abbiamo scelto di utilizzarli in questo libro.

Consideriamo il segmento di diagramma di flusso della struttura sequenziale, in Figura 2.1. Il *rettangolo*, detto anche *simbolo di azione*, indica qualsiasi tipo di operazione, come un calcolo o un input/output.

Le linee di flusso in figura indicano l'ordine in cui sono effettuate le operazioni: prima `grade` viene sommato a `total`, quindi viene sommato `1` a `counter`. Il C++ non limita il numero di azioni che si possono includere in una struttura sequenziale. Vedremo presto che ovunque sia possibile porre una sola azione, è anche possibile porre un intero blocco di azioni.

Se un diagramma di flusso deve rappresentare un algoritmo *completo*, si usa porre all'entrata del diagramma un simbolo *ovale* con la parola Begin (Inizio); analogamente all'uscita del diagramma si pone un ovale con la parola End (Fine). Se si disegna soltanto una parte dell'algoritmo generalmente si sostituiscono gli ovali con dei *cerchietti*, detti anche *simboli di connessione*.

Fra i simboli più importanti di un diagramma di flusso troviamo senz'altro il *rombo*, detto anche *simbolo decisionale*, il quale indica che deve essere presa una decisione. Torneremo a parlare di questo simbolo nella sezione seguente.

Il C++ prevede tre tipi di strutture di controllo che verranno presentate in questo capitolo. La struttura di selezione `if` compie un'operazione soltanto se una condizione è `true` (vera), mentre la salta se la condizione è `false` (falsa). La struttura di selezione `if/else` compie un'operazione se la condizione è `true`, mentre ne compie un'altra se la condizione è `false`. La struttura di selezione `switch` compie un'operazione scelta tra un'insieme di alternative, a seconda del valore di un'espressione.

Il costrutto `if` è una *struttura di selezione singola* perché sceglie o ignora una singola azione determinata. Il costrutto `if/else` è una *struttura di selezione doppia* perché sceglie una tra due azioni alternative. Il costrutto `switch` è una *struttura di selezione multipla* perché sceglie una tra diverse azioni alternative.

In C++ esistono tre tipi di strutture iterative, `while`, `do/while` e `for`. Per inciso ognuna delle parole `if`, `else`, `switch`, `while` e `do` è una *parola riservata*, cioè è una parola riservata, il cui compito è denotare una caratteristica ben precisa del linguaggio di programmazione. Non è possibile utilizzare una parola riservata come identificatore, per esempio per il nome di una variabile. La Figura 2.2 mostra l'elenco completo delle parole riservate del C++.

Parole riservate del C++

Parole riservate del C e del C++

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

Parole riservate del C++ ma non del C

`asm`	`bool`	`catch`	`class`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`false`	`friend`
`inline`	`mutable`	`namespace`	`new`	`operator`
`private`	`protected`	`public`	`reinterpret_cast`	
`static_cast`	`template`	`this`	`throw`	`true`
`try`	`typeid`	`typename`	`using`	`virtual`
`wchar_t`				

Figura 2.2 Parole riservate del C++.

Errore comune 2.1

Se utilizzate una parola riservata come identificatore commettete un errore di sintassi.

In un certo senso il discorso si esaurisce qui: in C++ esistono soltanto sette strutture di controllo, ovvero la struttura sequenziale, tre tipi di strutture di selezione e tre tipi di strutture di iterazione. Qualsiasi programma è formato unicamente da queste, naturalmente in tutte le possibili combinazioni, a seconda dell'algoritmo alla base del programma. Come per la struttura sequenziale di Figura 2.1, vedremo che in un diagramma di flusso ogni struttura di controllo contiene due cerchietti, uno all'entrata e un altro all'uscita di essa. Le strutture *a una sola entrata / a una sola uscita* semplificano la scrittura dei programmi, perché basta congiungere le varie strutture sovrapponendo l'entrata di una all'uscita della precedente. Il risultato di queste sovrapposizioni prende il nome di *sequenza di strutture di controllo*. Esiste un solo altro metodo per collegare tra loro diverse strutture di controllo: la *nidificazione*.

Ingegneria del software 2.1

Qualsiasi programma in C++ si può costruire a partire da sole sette strutture di controllo (`if`, `if/else`, `switch`, `while`, `do/while` e `for`), combinandole in due possibili modi, ovvero mettendole in sequenza o nidificandole.

2.5 La struttura di selezione if

Una struttura di selezione serve a decidere quale azione intraprendere tra diverse alternative possibili. Per esempio, supponiamo che il punteggio minimo per superare un esame sia 60. Lo pseudocodice

Se il voto dello studente è maggiore o uguale a 60
    Visualizza "Promosso"

valuta se la condizione "voto dello studente maggiore o uguale a 60" è vera o falsa. Se la condizione è vera viene visualizzato "Promosso", e viene eseguita l'istruzione seguente. Se la condizione è falsa, l'istruzione di visualizzazione viene saltata, e si passa direttamente all'istruzione successiva del programma. Avrete notato la spaziatura di rientro (indentazione) che abbiamo utilizzato per la seconda linea della struttura di selezione: in realtà è opzionale, ma vogliamo incoraggiarvi a usarla perché evidenzia visualmente la struttura di un programma strutturato. Quando convertirete il vostro pseudocodice in codice C++, il compilatore ignorerà gli spazi vuoti, le tabulazioni e gli a capo dei rientri presenti nel testo del programma.

Buona abitudine 2.1

Se applicate una spaziatura coerente e intelligente migliorerete notevolmente la leggibilità dei vostri programmi. Vi suggeriamo un rientro di tre spazi o di un carattere di tabulazione per ogni livello di rientro.

Lo pseudocodice precedente si può tradurre in C++ in questo modo:

```
if ( grade >= 60 )
   cout << "Passed";
```

In questo caso il codice C++ corrisponde quasi esattamente allo pseudocodice: è per questo che riteniamo lo pseudocodice un valido strumento per sviluppare i programmi.

Buona abitudine 2.2

Lo pseudocodice viene utilizzato per mettere nero su bianco le varie idee che possono venire in mente durante la fase di progettazione di un programma. La conversione successiva di pseudocodice in programma C++ è un'operazione semplice e diretta.

Il diagramma di flusso in Figura 2.3 illustra la struttura di selezione singola `if`. Il diagramma contiene un importante simbolo, il *rombo*, il simbolo delle decisioni, che indica che deve essere presa una decisione. Il simbolo contiene al suo interno un'espressione condizionale che può essere vera o falsa. Dal simbolo si diramano due linee di flusso: una indica la direzione che verrà presa se la condizione è vera, l'altra indica la direzione alternativa nel caso la condizione sia falsa. Nel Capitolo 1 abbiamo detto che una decisione viene presa sulla base di un'espressione condizionale, che contiene operatori relazionali o di uguaglianza. Qui vogliamo estendere questo concetto, e diciamo che una decisione può essere presa sulla base di qualsiasi espressione, con la convenzione che se il valore dell'espressione è zero si assume il significato `false`, mentre per qualsiasi altro valore il suo significato è `true`. Lo standard del C++ prevede un tipo di dato apposito, `bool`, per rappresentare i due valori `true` e `false`. Le parole riservate `true` e `false` sono gli unici due valori possibili di questo tipo di dato.

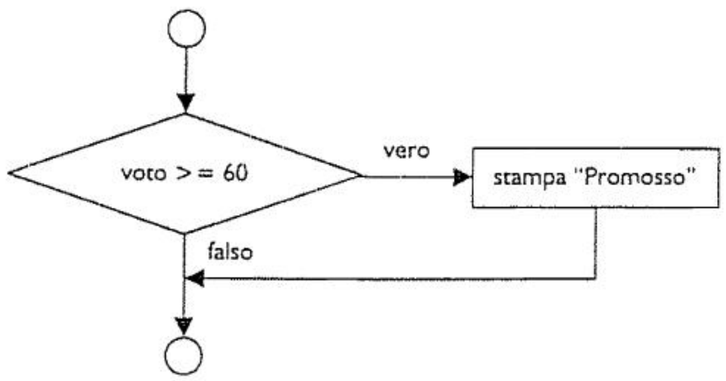


Figura 2.3 Il diagramma di flusso della struttura di selezione singola if.

Anche il costrutto `if` è una struttura a una sola entrata / una sola uscita. Come vedremo, anche i diagrammi di flusso delle altre strutture di controllo contengono rettangoli per le operazioni, e rombi per le decisioni, oltre naturalmente a cerchietti e linee di flusso. Questo costituisce il modello di programmazione ad *azioni/decisioni* su cui ponevamo l'accento all'inizio del capitolo. Un programma di qualsiasi complessità è formato da una combinazione delle sette strutture di controllo disponibili. In fase di progettazione immaginatevele vuote: nei rettangoli o nei rombi non è scritto nulla. Il vostro compito di programmatori consiste nell'assemblare tante strutture quante ne richiede l'algoritmo, con le due modalità possibili (pila o nidificazione), e nel riempire successivamente i vari simboli con le azioni e le decisioni appropriate.

2.6 La struttura di selezione if/else

La struttura `if` esegue l'azione indicata solamente se la condizione è vera, altrimenti la salta. La struttura `if/else` consente al programmatore di indicare un'azione alternativa, nel caso la condizione si falsa: quindi se la condizione risulta vera viene eseguita una data azione, se invece risulta falsa ne viene eseguita un'altra.

Per esempio, l'istruzione in pseudocodice

Se il voto dello studente è maggiore o uguale a 60
    Visualizza "Promosso"
altrimenti
    Visualizza "Bocciato"

visualizza *Promosso* se il voto dello studente è maggiore o uguale a 60, *Bocciato* se il voto non raggiunge 60. In ogni caso, dopo la visualizzazione del messaggio, viene eseguita l'istruzione che segue la struttura di selezione. Come vedete, abbiamo rientrato anche il corpo della *else*.

Buona abitudine 2.3

È buona norma rientrare entrambe le istruzioni alternative di un costrutto if/else.

Indipendentemente dalla convenzione che scegliete per il rientro del testo, cercate di applicarla in modo uniforme nei vostri programmi. Se non lo fate, potreste addirittura peggiorarne la leggibilità.

Buona abitudine 2.4

Se utilizzate diversi livelli di rientro, ognuno di essi dovrebbe rientrare di un numero fisso di spaziature.

Lo pseudocodice che abbiamo appena visto si può tradurre in C++ in questo modo:

```
if ( grade >= 60 )
   cout << "Passed";
else
   cout << "Failed";
```

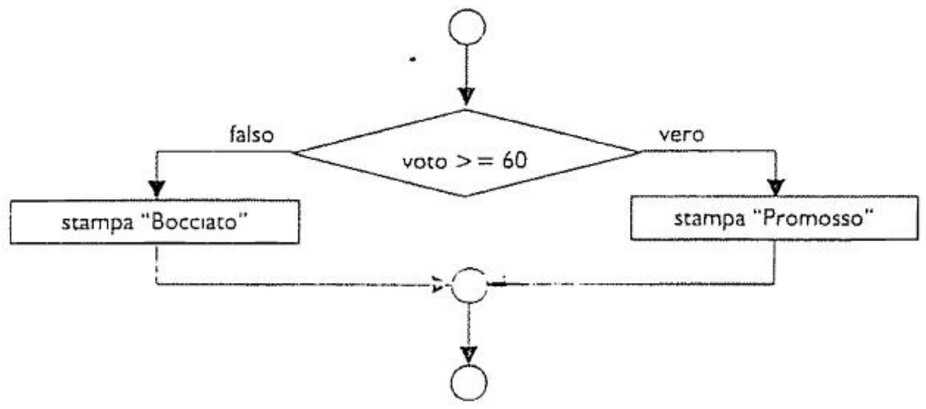


Figura 2.4 Diagramma di flusso della struttura di selezione doppia if/else.

In C++ esiste l'*operatore condizionale* (`?:`) che dal punto di vista operativo è molto simile al costrutto `if/else`. L'operatore condizionale è l'unico operatore *ternario* del C++, cioè lavora su tre operandi. Operandi e operatore formano una *espressione condizionale*: il primo operando costituisce la condizione, il secondo operando è il valore che assumerà l'intera espressione se la condizione è vera, mentre il terzo è il valore che assumerà l'intera espressione se la condizione è falsa. Osservate questa istruzione di output:

```
cout << ( grade >= 60 ? "Passed"   "Failed" );
```

Essa contiene un'espressione condizionale, il cui valore è `"Passed"` se la condizione `grade >= 60` è vera, mentre è `"Failed"` se la condizione è falsa. Come vedete un operatore condizionale equivale a un costrutto `if/else`. La precedenza dell'operatore condizionale è bassa, ed è per questo che nell'espressione precedente abbiamo dovuto inserire una coppia di parentesi. Il valore di un'espressione condizionale può anche consistere in azioni da eseguire. Per esempio l'espressione seguente

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

va letta "Se `grade` è maggiore o uguale a `60` allora `cout << "Passed"`, altrimenti `cout << "Failed"`. Anche questa operazione avrebbe potuto essere effettuata in modo equivalente da un costrutto `if/else`. Vedremo però che gli operatori condizionali possono essere utilizzati in alcune situazioni in cui non è consentito l'utilizzo di costrutti `if/else`.

I costrutti `if/else` *nidificati* servono a valutare una serie di casi e si formano inserendo costrutti `if/else` all'interno di altri costrutti `if/else`. Per esempio lo pseudocodice seguente stampa `A` se il punteggio dell'esame è maggiore o uguale a 90, `B` se è compreso tra 80 e 89, `C` se è compreso tra 70 e 79, `D` se è compreso tra 60 e 69 e `F` se è minore di 60.

```
Se il voto dello studente è maggiore o uguale a 90
      Visualizza "A"
altrimenti
      Se il voto dello studente è maggiore o uguale a 80
            Visualizza "B"
      altrimenti
            Se il voto dello studente è maggiore o uguale a 70
                  Visualizza "C"
            altrimenti
                  Se il voto dello studente è maggiore o uguale a 60
                        Visualizza "D"
                  altrimenti
                        Visualizza "F"
```

La versione in C++ di questo pseudocodice è:

```
if ( grade >= 90 )
   cout << "A";
else
   if ( grade >= 80 )
      cout << "B";
   else
      if ( grade >= 70 )
         cout << "C";
      else
         if ( grade >= 60 )
```



```
         cout << "D";
      else
         cout << "F";
```

Se grade è maggiore o uguale a 90 sono vere tutte le prime quattro condizioni, ma l'unica istruzione eseguita è la `cout` che segue il primo test. Dopo di che, la porzione `else` del costrutto `if/else` più esterno viene saltata. Molti programmatori adottano uno stile diverso per questo codice:

```
if ( grade >= 90 )
   cout << "A";
else if ( grade >= 80 )
   cout << "B";
else if ( grade >= 70 )
   cout << "C";
else if ( grade >= 60 )
   cout << "D";
else
   cout << "F";
```

ma naturalmente le due forme sono equivalenti. Molti preferiscono la seconda, perché il rientro non raggiunge un livello troppo marcato: se si sceglie la prima forma, invece, nelle ultime linee rimane poco spazio a disposizione per scrivere le istruzioni, e se si interrompono le linee andando a capo si può anche peggiorare la leggibilità del programma.

Obiettivo efficienza 2.1

Un costrutto `if/else` nidificato può essere molto più veloce di una serie di costrutti `if` in sequenza, perché una volta trovata la condizione vera, le altre sono tralasciate.

Obiettivo efficienza 2.2

In un costrutto `if/else` nidificato conviene verificare per prime le condizioni che hanno maggiore probabilità di risultare vere: in questo modo l'esecuzione sarà più rapida, perché l'esecuzione lascerà prima il costrutto `if/else`, non appena avrà trovato appunto la prima condizione vera.

La struttura di selezione `if` prevede soltanto un'istruzione nel proprio corpo. Per includerne più di una basta racchiuderle tutte tra una coppia di parentesi graffe (`{` e `}`). Un insieme di istruzioni racchiuse tra parentesi graffe si chiama *istruzione composta*.

Ingegneria del software 2.2

Un'istruzione composta può essere posta ovunque le regole del linguaggio permettano di scrivere un'istruzione singola.

L'esempio che segue contiene un'istruzione composta nella sezione `else` di un costrutto `if/else`:

```
if ( grade >= 60 )
   cout << "Passed.\n";
else {
   cout << "Failed.\n";
   cout << "You must take this course again.\n";
}
```

In questo caso se `grade` è minore di 60 il programma esegue entrambe le istruzioni presenti nel corpo della `else` e visualizza

```
Failed.
You must take this course again.
```

Notate le parentesi che racchiudono le due istruzioni della `else`. Queste sono di estrema importanza: infatti se le omettessimo l'istruzione

```
cout << "You must take this course again.\n";
```

si troverebbe al di fuori del corpo della `else` e sarebbe eseguita comunque, indipendentemente dalla condizione verificata da `if`.

Errore comune 2.2

Se dimenticate di scrivere una o entrambe le parentesi graffe che delimitano un'istruzione composta commetterete errori di sintassi o logici.

Buona abitudine 2.5

Se scrivete sempre le parentesi graffe nei costrutti `if`/`else` *o nelle altre strutture di controllo avrete meno probabilità di dimenticarle quando sono indispensabili. Ciò può capitarvi specialmente se aggiungete in un secondo tempo delle istruzioni aggiuntive in clausole* `if` *o* `else` *già esistenti.*

Il compilatore è in grado di segnalarvi solo gli errori di sintassi presenti nel codice del programma. Un errore logico invece si rivela soltanto durante l'esecuzione del programma. Un errore logico fatale termina immediatamente l'esecuzione, mentre un errore logico non fatale consente al programma di portare a termine il suo compito, ma i risultati prodotti saranno quasi certamente errati.

Ingegneria del software 2.3

Abbiamo visto che è possibile porre un'istruzione composta ovunque sia prevista la presenza di un'istruzione semplice: allo stesso modo si può porre un'istruzione nulla ovunque sia presente un'istruzione semplice. Un'istruzione nulla è formata semplicemente da un punto e virgola (;).

Errore comune 2.3

Se scrivete un punto e virgola dopo la condizione di un costrutto `if` *commettete un errore logico nel caso di un costrutto* `if` *a selezione singola, e un errore di sintassi nel caso di un costrutto* `if` *a selezione doppia (se la porzione* `if` *contiene effettivamente un'istruzione nel suo corpo).*

Buona abitudine 2.6

Alcuni programmatori preferiscono digitare subito le due parentesi graffe di un'istruzione composta, prima delle istruzioni effettive. In questo modo evitano di dimenticarne una o entrambe.

In questa sezione abbiamo introdotto il concetto di istruzione composta. In un'istruzione composta potete inserire anche delle dichiarazioni, come accade per esempio nel corpo di `main`. Un'istruzione composta che contiene delle dichiarazioni prende il nome di

blocco. Generalmente in un blocco le dichiarazioni precedono le istruzioni, ma è possibile porle ovunque all'interno del blocco, purché si trovino prima che le variabili siano effettivamente utilizzate.

Approfondiremo le caratteristiche dei blocchi nel Capitolo 3: fino ad allora evitate di utilizzarli se non sono strettamente necessari (come in `main`).

2.7 La struttura iterativa while

Una *struttura iterativa* consente di eseguire ripetutamente un'azione specifica fin tanto che una data condizione rimane vera. L'istruzione in pseudocodice

Finché ci sono ancora cose da comprare
    devo comprare il prossimo oggetto e depennarlo dalla lista

descrive l'azione ripetuta propria di una assidua sessione di shopping.

La condizione "ci sono altre cose da comprare" può essere vera o falsa. Se è vera si effettua l'operazione "devo comprare il prossimo oggetto e depennarlo dalla lista". L'operazione sarà eseguita continuamente, finché la condizione rimarrà vera. L'istruzione contenuta in un costrutto `while` costituisce il corpo del `while`. Naturalmente anche l'istruzione singola prevista dalla `while` può essere sostituita da un'istruzione composta.

A un certo punto la condizione diventerà falsa (nel nostro caso quando non ci sono più cose da comprare): in quel momento l'iterazione termina, e l'esecuzione del programma prosegue dalla prima istruzione che segue il costrutto `while`.

Errore comune 2.4

Se l'operazione all'interno della `while` *non è tale da rendere la condizione falsa a un certo punto, si ha un ciclo infinito, perché l'iterazione non ha mai modo di terminare.*

Errore comune 2.5

Fate attenzione a maiuscole e minuscole: se scrivete `While` *al posto di* `while` *commettete un errore di sintassi, perché il C++ fa differenza tra lettere maiuscole e minuscole. Tutte le parole riservate del C++ contengono solo lettere minuscole.*

Utilizziamo `while` in un problema reale, per cercare la prima potenza di 2 superiore a 1000. Supponiamo di aver inizializzato la variabile intera `product` a 2. Al termine del costrutto `while`, `product` conterrà il valore che stiamo cercando:

```
int product = 2;
while ( product <= 1000 )
   product = 2 * product;
```

Il diagramma di flusso di Figura 2.5 mostra il flusso di controllo del nostro programma. Anche qui sono presenti solo dei rettangoli e un rombo.

Il diagramma visualizza chiaramente la ripetizione: la linea di flusso che parte dal rettangolo torna sulla condizione, finché essa ad un certo punto risulta falsa. A quel punto il programma esce dal costrutto `while` ed esegue l'istruzione seguente del listato.

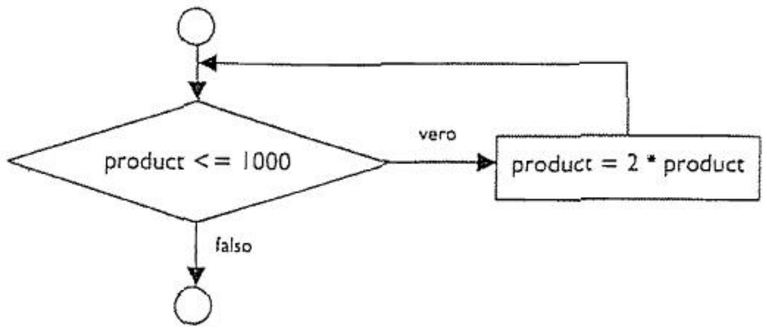


Figura 2.5 Diagramma di flusso della struttura di iterazione while.

La prima volta che si entra in `while` il valore di `product` è 2. La variabile viene poi moltiplicata ripetutamente per 2, assumendo i valori 4, 8, 16, 32, 64, 128, 256, 512 e 1024. Quando `product` assume il valore 1024 la condizione `product <= 1000` diventa falsa: ciò termina l'iterazione, e il valore finale di `product` resta 1024. L'esecuzione del programma riprende dall'istruzione che si trova subito dopo il costrutto `while`.

2.8 Tipologie degli algoritmi di iterazione: l'iterazione controllata da un contatore

Esistono diverse tipologie di algoritmi che sfruttano le strutture iterative. Nelle prossime sezioni ve ne mostreremo alcune, cercando di risolvere diverse varianti di uno stesso problema, e cioè il calcolo della media dei voti ricevuti dagli studenti di una classe. Ecco la definizione del nostro problema:

Una classe di dieci studenti ha sostenuto un esame. Avete a disposizione i voti di ogni studente, in una scala da 0 a 100. Determinate la media dei voti.

La media dei voti è uguale alla somma dei voti diviso il numero degli studenti. L'algoritmo di questo problema deve prevedere l'input dei dieci voti, il calcolo effettivo della media e l'output del risultato.

Scegliamo di utilizzare ancora una volta lo pseudocodice per fissare l'ordine di esecuzione delle varie operazioni. Ci serviremo della *iterazione controllata da un contatore* per ricevere in input i dieci voti. Questa tecnica fa uso di una variabile detta *contatore* che controlla quante volte viene eseguito un'insieme di istruzioni. Nel nostro esempio la ripetizione termina quando il contatore supera il valore 10, perché vogliamo effettuare l'input di solo 10 voti. L'algoritmo in pseudocodice è in Figura 2.6, e il programma corrispondente è in Figura 2.7. Nella prossima sezione vedremo come sono sviluppati gli algoritmi in pseudocodice. La ripetizione controllata da un contatore viene anche detta *ripetizione definita* perché il numero di ripetizioni è già noto prima dell'inizio del ciclo.

Nell'algoritmo ci sono riferimenti a un *totale* e a un *contatore*. Un *totale* è una variabile che serve ad accumulare progressivamente la somma di una serie di valori. Un *contatore* è una variabile che serve a effettuare un conteggio: in questo caso sarà utile per contare il numero dei voti già ricevuti in input in un dato momento. Una variabile totale deve essere inizializzata a zero prima di essere utilizzata, altrimenti la somma finale includerà il valore che si trova casualmente nella locazione di memoria che le è stata assegnata.

Inizializzare il totale a zero
Inizializzare il contatore dei voti a uno

Finché il contatore resta minore o uguale a dieci
    Prendere dall'input il prossimo voto
    Aggiungere il voto al totale
    Aggiungere uno al contatore

Impostare il valore della media al totale diviso dieci
Visualizzare la media

Figura 2.6 Algoritmo in pseudocodice per risolvere il problema della media dei voti di una classe con l'iterazione controllata da un contatore.

```
1   // Fig. 2.7: fig02_07.cpp   Media dei voti di una classe
2   // con l'iterazione controllata da un contatore
3   #include <iostream.h>
4
5   int main()
6   {
7      int total,        // somma dei voti
8          gradeCounter, // numero di voti immessi
9          grade,        // un voto
10         average;      // media dei voti
11
12     // fase di inizializzazione
13     total = 0;                          // azzera il totale
14     gradeCounter = 1;                   // prepara il ciclo
15
16     // fase di elaborazione
17     while ( gradeCounter <= 10 ) {      // ciclo 10 volte
18        cout << "Enter grade: ";         // prompt di input
19        cin >> grade;                    // input del voto
20        total = total + grade;  // aggiunge il voto al totale
21        gradeCounter=gradeCounter+1;//incrementa il contatore
22     }
23
24     // fase terminale
25     average = total / 10;               // divisione intera
26     cout << "Class average is " << average << endl;
27
28     return 0;   // il programma è terminato con successo
29  }
```

Figura 2.7 Programma in C++ con relativo output per il problema della media dei voti di una classe, risolto con l'iterazione controllata da un contatore (continua)

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Figura 2.7 Programma in C++ con relativo output per il problema della media dei voti di una classe, risolto con l'iterazione controllata da un contatore.

I contatori sono inizializzati generalmente a zero o a uno, a seconda di come si intende utilizzarli: noi presenteremo esempi di entrambi i casi.

Errore comune 2.6

Se dimenticate di inizializzare un contatore, il risultato del vostro programma sarà quasi certamente errato. Questo è un esempio di errore logico.

Buona abitudine 2.7

Inizializzate sempre contatori e totali.

Buona abitudine 2.8

Dichiarate ogni variabile su una linea diversa.

Se osservate, la media calcolata dal nostro programma è un numero intero. In realtà la somma dei voti nel nostro esempio è 817, che diviso 10 dà 81.7, un numero che ha una parte decimale. Nella prossima sezione impareremo ad utilizzare questo tipo di numeri, detti valori a virgola mobile.

Errore comune 2.7

Il contatore di un ciclo assume un valore maggiore di 1 rispetto all'ultimo valore valido (per es. nel caso del conteggio da 1 a 10 il contatore assume per ultimo il valore 11): se riutilizzate il valore del contatore per altri calcoli dopo la fine del ciclo potreste commettere un errore.

In Figura 2.7, se la linea 25 utilizzasse `gradeCounter` anziché 10 per il calcolo, il programma visualizzerebbe in output il valore 74.

2.9 Tipologie degli algoritmi di iterazione: l'iterazione controllata da un valore sentinella

Adesso proviamo a generalizzare il problema della media del paragrafo precedente. Ecco la definizione del nostro nuovo problema:

Scrivete un programma che calcoli la media dei voti di un numero arbitrario di studenti: il numero di studenti verrà deciso volta per volta dall'utente durante l'esecuzione.

Nell'esempio precedente sapevamo dall'inizio che i voti in input sarebbero stati 10. Qui invece non abbiamo idea di quanti saranno. Come può determinare il programma quando fermarsi nel richiedere i voti in input?

Una soluzione si serve di un valore speciale, detto *valore sentinella*, *valore segnale*, o *flag*: il suo scopo è indicare la fine dell'input dei dati. Durante l'esecuzione l'utente immette tutti i voti che desidera, quindi scrive il valore sentinella, indicando che i voti sono finiti. Le iterazioni controllate da valori sentinella sono *ripetizioni indefinite*, perché prima dell'esecuzione non si può sapere quante volte sarà eseguito il ciclo. È chiaro che occorre scegliere il valore sentinella in modo attento, perche non deve essere confuso con un voto valido. Dato che i voti di un test sono generalmente numeri non negativi possiamo scegliere −1 come valore sentinella per il nostro problema. Così l'utente può immettere per esempio i valori 95, 96, 74, 75, 89 e −1 durante l'esecuzione. La media sarà calcolata soltanto sui valori 95, 96, 74, 75 e 89: -1 non è un voto e quindi non entrera nel computo della media.

Errore comune 2.8

Non scegliete un valore sentinella che può essere anche un valore legittimo di input: si tratta di un errore logico.

Adesso cerchiamo di risolvere il problema della media con una tecnica nota come *ridefinizione attraverso passi di raffinamento top-down* (dall'alto in basso), che e di una certa utilità per creare programmi ben strutturati. Cominciamo a rappresentare in pseudocodice il *top*, cioè il livello più alto, o generico:

Determinare la media dei voti di una classe relativa per un determinato esame.

Il livello massimo consiste in una sola frase che indica il compito complessivo del programma e in questo senso è già una rappresentazione completa del programma. Purtroppo difficilmente riesce a descrivere una quantità di dettagli sufficiente a ricavare in modo immediato il programma in C++. È qui che comincia il processo di raffinamento: suddividiamo il top in una serie di compiti più semplici ed elenchiamoli nell'ordine in cui vanno eseguiti. Il *primo raffinamento* può essere questa:

Inizializzare le variabili
Effettuare l'input, la somma e il conteggio dei voti
Calcolare e visualizzare la media dei voti

Qui abbiamo utilizzato soltanto una struttura sequenziale, cioè i vari passi sono eseguiti nell'ordine dato, uno dopo l'altro.

Ingegneria del software 2.4

Ogni passo di raffinamento, incluso il livello top, *è una descrizione completa dell'algoritmo: l'unica variazione che si ha passando da un livello a un altro riguarda il livello di dettaglio.*

Ingegneria del software 2.5

È possibile suddividere i programmi in tre fasi logiche: una fase di inizializzazione, dove si inizializzano le variabili; una fase di elaborazione, dove si ricevono in input i dati e si regolano di conseguenza le variabili; infine una fase finale dove si calcolano e si visualizzano i risultati.

Fate riferimento all'ultimo consiglio di *Ingegneria del software* per cimentarvi con la vostra prima ridefinizione. Nel prossimo passo di raffinamento, dobbiamo stabilire di quali variabili abbiamo bisogno: occorrono una variabile totale, un contatore dei numeri già elaborati, una variabile che assuma il valore di ciascun voto durante l'input e una variabile che memorizzi il valore della media. L'istruzione in pseudocodice

Inizializzare le variabili

può essere ridefinita in:

Inizializzare il totale a zero
Inizializzare il contatore a zero

Come avrete notato si devono inizializzare solo *total* (totale) e *counter* (contatore) prima che possano essere utilizzate: *average* (per la media) e *grade* (per l'input dell'utente) non hanno bisogno di un'inizializzazione, perché i loro valori precedenti saranno sovrascritti durante il calcolo e l'input.

L'istruzione in pseudocodice

Effettuare l'input, la somma e il conteggio dei voti

richiede una struttura di iterazione, cioè un ciclo, per l'immissione di ciascun voto. Dato che non sappiamo in anticipo quanti saranno i voti, utilizzeremo un'iterazione controllata da un valore sentinella. L'utente immetterà i voti uno dopo l'altro: dopo l'ultimo valore legittimo immetterà il valore sentinella. Per ogni voto immesso il programma controllerà che non si tratti del valore sentinella: se è così terminerà la fase di input e procederà a calcolare la media. Ecco dunque la seconda ridefinizione in pseudocodice:

```
Effettuare l'input del primo voto (può essere il valore sentinella)
Finché l'utente non ha ancora digitato il valore sentinella
    Aggiungere il voto corrente al totale
    Aggiungere uno al contatore
    Effettuare l'input del prossimo voto (può essere il valore sentinella)
```

Naturalmente nello pseudocodice non ci prendiamo la briga di mettere tra parentesi graffe le istruzioni del corpo del `while`. Ci basta utilizzare un rientro per indicare che appartengono tutte alla *while*. Lo pseudocodice è soltanto uno strumento informale, non è il caso di diventare pedanti con la sintassi delle istruzioni.

L'istruzione

Calcolare e visualizzare la media dei voti

può essere ridefinita così:

```
Se il contatore non è uguale a zero
    Impostare il valore della media al totale diviso il contatore
    Visualizzare la media
altrimenti
    Visualizzare "Non è stato digitato alcun voto"
```

Abbiamo anche incluso un controllo per scongiurare un'eventuale divisione per zero: un'operazione di questo tipo sarebbe un errore fatale e farebbe terminare immediatamente il programma. La Figura 2.8 mostra completamente il secondo raffinamento in pseudocodice.


```
Inizializzare il totale a zero
Inizializzare il contatore a zero

Effettuare l'input del primo voto (può essere il valore sentinella)
Finché l'utente non ha ancora digitato il valore sentinella
    Aggiungere il voto corrente al totale
    Aggiungere uno al contatore
    Effettuare l'input del prossimo voto (può essere il valore sentinella)

Se il contatore non è uguale a zero
    Impostare il valore della media al totale diviso il contatore
    Visualizzare la media
altrimenti
    Visualizzare "Non è stato digitato alcun voto"
```

Figura 2.8 Algoritmo in pseudocodice che risolve il problema della media dei voti di una classe con l'iterazione controllata da un valore sentinella.

Errore comune 2.9

Il tentativo di dividere un numero per zero causa un errore fatale in fase di esecuzione.

Buona abitudine 2.9

Se effettuate una divisione per espressioni che potrebbero risultare nulle, prevedete esplicitamente un test di verifica su di esse prima di procedere. Se rilevate in anticipo una possibile divisione per zero prevedete anche un comportamento adeguato del vostro programma, come la visualizzazione di un avvertimento, anziché lasciar correre.

In Figura 2.6 e in Figura 2.8 abbiamo inserito qualche linea vuota nello pseudocodice per migliorarne la leggibilità. Così abbiamo evidenziato anche i passaggi tra le varie fasi del programma.

L'algoritmo in pseudocodice di Figura 2.8 risolve il problema più generale del calcolo della media di una classe di studenti. In questo caso sono bastati soltanto due livelli di ridefinizione, ma a volte ne sono necessari diversi altri.

Ingegneria del software 2.6

Potete terminare il processo di ridefinizione quando avete un livello di dettaglio abbastanza buono per poter iniziare la conversione da pseudocodice a C++. A questo punto l'implementazione del programma in C++ è semplice e immediata.

In Figura 2.9 vediamo il programma risultante e il suo output. Anche se immettiamo solo numeri interi, la media avrà con ogni probabilità una parte decimale. Il tipo `int` non è in grado di rappresentare i numeri reali. Per questo nel programma ci serviamo del tipo di dati `float`, che rappresenta numeri con parti decimali (in inglese *floating-point numbers*, numeri a virgola mobile). In questo programma utilizziamo per la prima volta anche un operatore speciale detto *operatore di cast*. Ci torneremo su dopo aver presentato il programma.

```
1   // Fig. 2.9: fig02_09.cpp
2   // Calcolo della media dei voti di una classe con
3   // l'iterazione controllata da un valore sentinella.
4   #include <iostream.h>
5   #include <iomanip.h>
6
7   int main()
8   {
9      int total,        // somma dei voti
10         gradeCounter, // numero di voti immessi
11         grade;        // un voto
12     float average; // numero con punto decimale per la media
13
14     // fase di inizializzazione
15     total = 0;
16     gradeCounter = 0;
17
18     // fase di elaborazione
19    cout << "Enter grade, -1 to end: ";
20     cin >> grade;
21
22     while ( grade != -1 ) {
23        total = total + grade;
24        gradeCounter = gradeCounter + 1;
25        cout << "Enter grade, -1 to end: ";
26        cin >> grade;
27     }
28
29     // fase terminale
30     if ( gradeCounter != 0 ) {
31        average = static_cast< float >(total) / gradeCounter;
32        cout << "Class average is " << setprecision( 2 )
33             << setiosflags( ios::fixed | ios::showpoint )
34             << average << endl;
35     }
36     else
37        cout << "No grades were entered" << endl;
38
39     return 0;   // il programma è terminato con successo
40  }
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
```

Figura 2.9 Programma in C++ e relativo output per il calcolo della media dei voti di una classe con l'iterazione controllata da un valore sentinella (continua)


```
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Figura 2.9 Programma in C++ e relativo output per il calcolo della media dei voti di una classe con l'iterazione controllata da un valore sentinella.

Notate l'istruzione composta in `while`. Senza le parentesi graffe le ultime tre istruzioni del corpo del ciclo si ritroverebbero al di fuori del ciclo. Il computer interpreterebbe questo come:

```
while ( grade != -1 )
   total = total + grade;
gradeCounter = gradeCounter + 1;
cout << "Enter grade, -1 to end: ";
cin >> grade;
```

Se l'utente non immette come voto iniziale il valore -1 si ha un ciclo senza fine.

Notate l'istruzione

```
cin >> grade;
```

L'istruzione che la precede avvisa l'utente che il programma è in attesa dei valori in input.

Buona abitudine 2.10

Avvisate l'utente quando il programma richiede l'input di valori. L'avviso, detto anche prompt, dovrebbe indicare il tipo di input richiesto e gli eventuali valori speciali ammessi (come il valore sentinella).

Buona abitudine 2.11

In un ciclo controllato da un valore sentinella è conveniente indicare o ricordare all'utente qual è tale valore.

Anche le medie aritmetiche effettuate su valori interi producono spesso numeri con parti decimali. Questi valori sono detti numeri reali, o a virgola mobile, e sono rappresentati dal tipo di dati `float`. La variabile `average` è dichiarata come `float`, in modo che la parte decimale del calcolo non sia troncata. Tuttavia l'operazione `total / gradeCounter` produce un intero, perché sia `total` che `gradeCounter` sono numeri interi.

La divisione di due numeri interi è una *divisione intera*, in cui le parti decimali vengono troncate comunque. Tale calcolo viene effettuato prima dell'assegnamento, per cui la parte decimale viene perduta prima di poter essere conservata in `average`. Per effettuare un calcolo reale con valori interi dobbiamo creare dei valori temporanei a virgola mobile. In C++ esiste l'*operatore di cast unario*, pensato appositamente per questo. L'istruzione

```
average = static_cast< float >( total ) / gradeCounter;
```

include l'operatore di cast `static_cast< float >()` che crea una copia temporanea in virgola mobile dell'operando tra parentesi, cioè `total`. Questa è una *conversione esplicita*. Il valore conservato in `total` è ancora intero, mentre il calcolo adesso viene effettuato tra un valore a virgola mobile (la versione temporanea `float` di `total`) e l'intero `gradeCounter`.

Il compilatore sa calcolare il valore delle espressioni soltanto se i tipi degli operandi sono identici. Per assicurare l'uguaglianza dei tipi, effettua preventivamente un'operazione detta *promozione*, o *conversione implicita*, sugli operandi selezionati. Per esempio, in un'espressione contenente valori `int` e `float`, tutti i valori `int` sono *promossi* a `float`. Nel nostro esempio il calcolo viene effettuato dopo la promozione di `gradeCounter` a `float`, e il risultato viene poi assegnato a `average`. Nel seguito del capitolo parleremo di tutti i tipi di dato standard e di come sono promossi.

Gli operatori di cast sono disponibili per tutti i tipi di dato. L'operatore `static_cast` si forma facendo seguire alla parola riservata `static_cast` due parentesi angolari che racchiudono il nome del tipo di dato. L'operatore di cast è *unario*, cioè prende un solo operando. Nel Capitolo 1 abbiamo visto operatori binari, gli operatori aritmetici. Il C++ prevede anche una versione unaria degli operatori più (+) e meno (-), che servono a dare il segno a dei valori numerici, come `-7` o `+5`.

Gli operatori di cast associano da destra a sinistra e, fra gli operatori unari, sono quelli che hanno precedenza più alta. La loro precedenza è anche più alta degli operatori *, / e %, ma è più bassa di quella delle parentesi. Nei nostri schemi di precedenza indichiamo l'operatore di cast con la notazione *`static_cast<type>()`*.

Le varianti di formattazione in Figura 2.9 sono discusse in dettaglio nel Capitolo 11: qui ne diamo soltanto dei brevi cenni. La chiamata a `setprecision(2)` in

```
cout << "Class average is " << setprecision( 2 )
     << setiosflags(ios::fixed | ios::showpoint)
     << average << endl;
```

indica che la variabile `average`, di tipo `float`, deve essere visualizzata con una *precisione* di due cifre, a destra del punto decimale: si avrà un output del tipo 92.37. La funzione chiamata è nota come *manipolatore di stream parametrizzato*. I programmi che usano queste funzioni devono contenere la seguente direttiva al preprocessore:

```
#include <iomanip.h>
```

Osservate che `endl` è un esempio di *manipolatore di stream non parametrizzato*, e non ha bisogno dell'inclusione del file di intestazione `iomanip.h`. Se non si indica che precisione utilizzare, i valori in virgola mobile sono inviati in output per default con sei cifre decimali, anche se c'è un'eccezione a questa regola che vedremo tra breve.

Il manipolatore di stream `setioflags(ios::fixed | ios::showpoint)` nell'istruzione precedente imposta due opzioni di formattazione dell'output, `ios::fixed` e `ios::showpoint`. Le barre verticali (|) separano le diverse opzioni presenti nella chiamata a `setioflags`: spiegheremo il significato di questa notazione nel Capitolo 5 del volume Tecniche Avanzate. L'opzione `ios::fixed` fa sì che il valore in virgola mobile sia inviato in output con una *formattazione a virgola fissa*: questa formattazione si contrappone alla *notazione scientifica*, che vedremo nel Capitolo 11. L'opzione `ios::showpoint` forza la visualizzazione del punto decimale e degli eventuali zero riempitivi, anche nel caso di un numero con parte decimale nulla, come 88.00. In mancanza di questa opzione non sarebbero visualizzati né il punto decimale né i due zeri che lo seguono. Con la precedente formattazione il valore visualizzato viene *arrotondato* al numero di posizioni decimali indicato, ma in memoria il suo valore rimane inalterato. Per esempio i valori 87.945 e 67.543 sono visualizzati rispettivamente come 87.95 e 67.54.

Errore comune 2.10

Se utilizzate numeri a virgola mobile non fate troppo affidamento sulla precisione della loro rappresentazione, o potreste avere dei risultati scorretti. Infatti la rappresentazione di un numero a virgola mobile, sulla maggior parte dei computer, è soltanto approssimativa.

Buona abitudine 2.12

A causa della loro rappresentazione approssimativa, non è opportuno verificare l'uguaglianza di due numeri a virgola mobile: conviene piuttosto verificare che la loro differenza in valore assoluto sia minore di una piccola quantità.

Sebbene i numeri a virgola mobile non siano pienamente affidabili dal punto di vista della precisione, possono essere usati in numerose applicazioni. Per esempio, se consideriamo la temperatura del corpo umano, possiamo accontentarci di una precisione a una sola cifra decimale (37.0 gradi Celsius). Se leggiamo 37.0 sul nostro termometro, il valore potrebbe essere in realtà 36.9997658578242. Ciò che conta è avere una buona precisione per il tipo di applicazione da sviluppare.

Anche la divisione produce spesso dei numeri a virgola mobile. Dividendo 10 per 3 avremo 3.3333333... con una serie infinita di 3 dopo il punto decimale. Il computer può allocare soltanto una quantità predefinita di spazio per memorizzare un numero, per cui l'approssimazione, in casi come questo, è un fatto inevitabile.

2.10 Tipologie di algoritmi e ridefinizione top-down: le strutture di controllo nidificate

Adesso proviamo ad analizzare la definizione di un altro problema. Utilizzeremo di nuovo lo pseudocodice e la ridefinizione top-down per giungere a scrivere il programma definitivo in C++. Abbiamo visto che le strutture di controllo possono essere messe in sequenza, una dopo l'altra. In questo paragrafo parleremo dell'unico modo alternativo di combinare le strutture di controllo, la *nidificazione*.

Eccovi la definizione del problema:

Un College ha istituito un corso di preparazione per l'esame di licenza di intermediario immobiliare. L'anno scorso un gran numero di studenti di questo corso ha affrontato l'esame finale. Naturalmente la direzione del College vuole conoscere l'andamento complessivo dell'esame, perciò vi chiede di scrivere un programma di riepilogo dei risultati. Vi è stata consegnata un elenco di 10 studenti, e in corrispondenza di ogni nome trovate la cifra 1 nel caso di promozione e la cifra 2 nel caso di bocciatura.

Il vostro programma dovrebbe:

1. Effettuare l'input di tutti i risultati (i valori possibili sono 1 e 2). Per richiedere ogni valore successivo dovrebbe visualizzare un messaggio.
2. Visualizzare un riepilogo dei risultati, che indichi il numero di studenti promossi e bocciati.
3. Se il numero di studenti promossi supera 8 visualizzare un messaggio che indica l'aumento della retta.

Rileggendo con attenzione la definizione del problema ci vengono spontanee le seguenti considerazioni:

1. Il programma deve elaborare 10 risultati. Possiamo utilizzare quindi un ciclo controllato da un contatore.
2. Ogni risultato è un numero, e può essere 1 o 2. Ogni volta che il programma legge un risultato deve verificare di quale valore si tratta. Nel nostro test verificheremo se il valore è 1: se non lo è, presumeremo che sia 2. Vedremo che conseguenze comporta questa assunzione in un esercizio a fine capitolo.
3. Utilizzeremo due contatori: uno per gli studenti che hanno superato l'esame e uno per gli studenti bocciati.
4. Al termine dell'input il programma deve decidere se sono stati promossi più di 8 studenti.

Procediamo adesso con il metodo di ridefinizione top-down. Il livello massimo (top) può essere:

Analizzare i risultati degli esami e decidere se si deve aumentare la retta.

È importante che questo livello contenga una rappresentazione completa del problema, anche se a un livello di dettaglio minimo. Valutiamo che saranno necessari anche parecchi livelli di ridefinizione prima di giungere al programma finale in C++. La nostra prima ridefinizione è questa:

Inizializzare le variabili
Effettuare l'input dei dieci voti, contando le promozioni e le bocciature
Visualizzare un riepilogo dei risultati e decidere se aumentare la retta

Come capite, abbiamo ancora bisogno di un ulteriore livello di ridefinizione, anche se la rappresentazione del programma è già completa. Facciamo mente locale sulle variabili da utilizzare: abbiamo bisogno di due contatori, uno per le promozioni e uno per le bocciature, di un contatore per il ciclo e di una variabile che contenga di volta in volta i valori immessi dall'utente. L'istruzione in pseudocodice

Inizializzare le variabili

può essere ridefinita così:

Inizializzare le promozioni a zero
Inizializzare le bocciature a zero
Inizializzare il contatore a uno

Come vedete solo i contatori e i totali sono inizializzati. L'istruzione in pseudocodice

Effettuare l'input dei dieci voti, contando le promozioni e le bocciature

è in realtà un ciclo che riceve in input i valori di tutti gli esami, uno dopo l'altro. Già prima di cominciare sappiamo che i valori sono in tutto 10, per cui possiamo tranquillamente utilizzare un ciclo controllato da contatore. All'interno del ciclo, *nidificata* in esso, troviamo una struttura di selezione doppia, che determina caso per caso se si tratta di una promozione o di una bocciatura e incrementa il contatore corrispondente. La ridefinizione dell'istruzione precedente è dunque:

Finché il contatore è minore o uguale a dieci
    Effettuare l'input del prossimo risultato

    Se lo studente è stato promosso
        Aggiungere uno alla variabile delle promozioni

    altrimenti
        Aggiungere uno alla variabile delle promozioni

    Aggiungere uno al contatore

Abbiamo inserito delle linee vuote per evidenziare la struttura di controllo `if/else`: così facendo abbiamo migliorato la leggibilità dello pseudocodice. L'istruzione

Visualizzare un riepilogo dei risultati e decidere se aumentare la retta

può essere ridefinita così:

Visualizzare il numero di promozioni
Visualizzare il numero di bocciature
Se sono stati promossi più di 8 studenti
    Visualizzare "La retta sarà aumentata "

La seconda ridefinizione completa è in Figura 2.10. Come avrete notato, abbiamo evidenziato con delle linee vuote anche la struttura di controllo `while`.

Inizializzare le promozioni a zero
Inizializzare le bocciature a zero
Inizializzare il contatore a uno

Finché il contatore è minore o uguale a dieci
    Effettuare l'input del prossimo risultato
    Se lo studente è stato promosso
        Aggiungere uno alla variabile delle promozioni
    altrimenti
        Aggiungere uno alla variabile delle promozioni
    Aggiungere uno al contatore

Visualizzare il numero di promozioni
Visualizzare il numero di bocciature
Se sono stati promossi più di 8 studenti
    Visualizzare "La retta sarà aumentata "

Figura 2.10 Pseudocodice dell'analisi dei risultati dell'esame.

A questo punto abbiamo un livello di raffinamento adeguato per poter passare a scrivere il programma vero e proprio: potete vederlo in Figura 2.11 insieme con il suo output. Nel programma abbiamo sfruttato una caratteristica del C++ che consente di inizializzare le variabili durante la loro dichiarazione. Se un programma contiene dei cicli di iterazione, generalmente queste inizializzazioni sono effettuate all'inizio di ogni ciclo: in altri casi si troverebbero in normali istruzioni di assegnamento.

```
1  // Fig. 2.11: fig02_11.cpp
2  // Analisi dei risultati di un esame
3  #include <iostream.h>
4
5  int main()
```

Figura 2.11 Programma in C++ per l'analisi dei risultati dell'esame (continua)

```
6   {
7      // inizializza le variabili nelle dichiarazioni
8      int passes = 0,            // numero di promossi
9          failures = 0,          // numero di bocciati
10         studentCounter = 1,    // contatore studenti
11         result;                // un risultato
12     //  elaborazione per 10 studenti;
13     //  ciclo controllato da un contatore
14     while ( studentCounter <= 10 ) {
15        cout << "Enter result (1=pass,2=fail): ";
16        cin >> result;
17
18        if ( result == 1 )      // if/else nidificato in while
19           passes = passes + 1;
20        else
21           failures = failures + 1;
22
23        studentCounter = studentCounter + 1;
24     }
25
26     // fase terminale
27     cout << "Passed " << passes << endl;
28     cout << "Failed " << failures << endl;
29
30     if ( passes > 8 )
31        cout << "Raise tuition " << endl;
32
33     return 0;  // indica che il programma e terminato con successo
34  }
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Figura 2.11 Programma in C++ per l'analisi dei risultati dell'esame (continua)


```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Figura 2.11 Programma in C++ e relativo output per l'analisi dei risultati dell'esame.

Buona abitudine 2.13

Se inizializzate le variabili nello stesso momento in cui le dichiarate eviterete il problema dei dati non inizializzati.

Ingegneria del software 2.7

L'esperienza dimostra che la parte più impegnativa della soluzione di un problema è lo sviluppo dell'algoritmo risolutivo. Una volta sviluppato un buon algoritmo, la scrittura del programma in C++ corrispondente è semplice e diretta.

Ingegneria del software 2.8

Molti programmatori esperti non avvertono la necessità di scrivere il loro programma prima in pseudocodice: essi credono che il fine ultimo del loro lavoro sia di avere tra le mani un programma completo e funzionante, e quindi vedono lo pseudocodice come una perdita di tempo che non fa altro che allontanare il loro obiettivo. Se si tratta di un'abitudine innocua per problemi di modesta complessità, nel caso di progetti complessi può comportare seri ritardi, specialmente se l'algoritmo ideato non è di buona qualità.

2.11 Gli operatori di assegnamento

In C++ esistono diversi operatori che permettono di abbreviare le espressioni di assegnamento. Per esempio l'istruzione

```
c = c + 3;
```

può essere abbreviata grazie all'*operatore di assegnamento con addizione* `+=` in

```
c += 3;
```

L'operatore `+=` somma il valore dell'espressione alla sua destra al valore della variabile alla sua sinistra e memorizza il risultato nella variabile. Qualsiasi istruzione nella forma

variabile = variabile operatore espressione;

dove *operatore* è un operatore binario a scelta tra `+`, `-`, `*`, `/` e `%` (insieme con altri di cui parleremo in seguito), può essere scritta nella forma

variabile operatore = espressione;

Questo significa che l'assegnamento `c += 3` incrementa `c` di 3. La Figura 2.12 elenca gli operatori di assegnamento aritmetici e mostra alcune espressioni in cui sono utilizzati.

Obiettivo efficienza 2.3

Grazie agli operatori di assegnamento aritmetici è possibile scrivere e compilare i programmi più velocemente. Alcuni compilatori, infatti, generano del codice eseguibile più efficiente in corrispondenza di tali operatori.

Obiettivo efficienza 2.4

Molti dei nostri consigli di questa sezione sono efficaci in teoria, ma se volete potete tranquillamente ignorarli. Troverete un miglioramento rilevante dell'efficienza se li applicate in segmenti critici del programma, per esempio nei cicli di iterazione.

Operatore	Esempio	Significato	Assegna
Per ipotesi: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c+= 7`	`c = c + 7`	10 a c
`-=`	`d -= 4`	`d = d - 4`	1 a d
`*=`	`e *= 5`	`e = e * 5`	20 a e
`/=`	`f /= 3`	`f = f / 3`	2 a f
`%=`	`g %= 9`	`g = g % 9`	3 a g

Figura 2.12 Gli operatori aritmetici di assegnamento.

2.12 Gli operatori di incremento e decremento

In C++ troviamo anche gli operatori unari di *incremento* (++) e *decremento* (--), presentati in Figura 2.13. Se la variabile `c` deve essere incrementata di 1 possiamo utilizzare l'operatore di incremento unario ++, anziché le espressioni equivalenti `c = c + 1` o `c += 1`. Se l'operatore di incremento o decremento viene scritto prima della variabile prende il nome di operatore di *preincremento* o *predecremento*. Se, al contrario, viene posposto a essa, prende il nome di operatore di *postincremento* o *postdecremento*. Se preincrementate una variabile, questa viene incrementata di 1, e nell'espressione in cui appare la variabile viene utilizzato il suo nuovo valore. Se postincrementate una variabile, nell'espressione in cui appare la variabile viene utilizzato il suo vecchio valore, e soltanto dopo la variabile viene incrementata di 1. La stessa cosa vale per il postincremento e il postdecremento.

Operatore	Nome	Esempio	Spiegazione
`++`	preincremento	`++a`	Incrementa `a` di 1, poi usa il nuovo valore di `a` nell'espressione in cui `a` si trova.
`++`	postincremento	`a++`	Usa il valore corrente di `a` nell'espressione in cui `a` si trova, poi incrementa `a` di 1.
`--`	predecremento	`--b`	Decrementa `b` di 1, poi usa il nuovo valore di `b` nell'espressione in cui `b` si trova.
`--`	postdecremento	`b--`	Usa il valore corrente di `b` nell'espressione in cui `b` si trova, poi decrementa `b` di 1.

Figura 2.13 Gli operatori di incremento e decremento.

Il programma in Figura 2.14 illustra la differenza tra il preincremento e il postincremento. Nel postincremento, la variabile `c` viene prima inviata in output e poi incrementata, mentre nel preincremento l'incremento ha luogo prima di ogni altra operazione.

Il programma visualizza il valore di `c` prima e dopo l'utilizzo dell'operatore ++. L'operatore di decremento -- ha un uso analogo.

```
1  // Fig. 2.14: fig02_14.cpp
2  // Preincremento e postincremento
3  #include <iostream.h>
4
5  int main()
6  {
7     int c;
8
9     c = 5;
10    cout << c << endl;          // visualizza 5
11    cout << c++ << endl;        // visualizza 5 e postincrementa
12    cout << c << endl << endl; // visualizza 6
13
14    c = 5;
15    cout << c << endl;          // visualizza 5
16    cout << ++c << endl;        // preincrementa e visualizza 6
17    cout << c << endl;          // visualizza 6
18
19    return 0;  // indica che il programma è terminato con successo
20 }
```

```
5
5
6

5
6
6
```

Figura 2.14 Differenza tra preincremento e postincremento.

Buona abitudine 2.14

Non scrivete alcuno spazio tra un operatore unario e il suo operando.

Le tre istruzioni di assegnamento di Figura 2.11

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

possono essere scritte in maniera più concisa con gli operatori aritmetici di assegnamento:

```
passes += 1;
failures += 1;
student += 1;
```

oppure con gli operatori di preincremento:

```
++passes;
++failures;
++student;
```

o gli operatori di postincremento:

```
passes++;
failures++;
student++;
```

Quando l'incremento o il decremento di una variabile sono le uniche operazioni di un'espressione, le forme di preincremento e postincremento sono equivalenti. Solo quando una variabile appare nel contesto di un'espressione più complessa l'utilizzo di preincremento o postincremento può influenzare il valore dell'espressione.

Per ora possiamo utilizzare come operando di un operatore di incremento o decremento soltanto un semplice nome di variabile: vedremo in seguito che questi operatori si devono utilizzare con i cosiddetti *lvalue*.

Errore comune 2.11

Se utilizzate un operatore di incremento o decremento su un'espressione diversa dal nome di una variabile, come ad esempio in `++(x + 1)`*, commettete un errore di sintassi.*

La Figura 2.15 illustra la precedenza e l'associatività di ogni operatore di cui abbiamo parlato finora. Gli operatori sono elencati in ordine decrescente di precedenza. La seconda colonna indica l'associatività degli operatori, per ogni livello di precedenza. Associano da destra a sinistra: l'operatore condizionale (`?:`), l'operatore unario di incremento (`++`) e decremento (`--`), l'operatore più (`+`), l'operatore meno (`-`), i cast e gli operatori di assegnamento `=`, `+=`, `-=`, `*=`, `/=` e `%=`. Tutti gli altri associano da sinistra a destra. La terza colonna della tabella indica il nome dei vari gruppi di operatori.

Operatori	Associatività	Tipo
`( )`	sx verso dx	parentesi
`++ -- + static_cast<`*type*`>()`	dx verso sx	unario
`* / %`	sx verso dx	moltiplicativo
`+ -`	sx verso dx	additivo
`<< >>`	sx verso dx	inserzione/estrazione
`< <= > >=`	sx verso dx	relazionale
`== !=`	sx verso dx	di uguaglianza
`?:`	dx verso sx	condizionale
`= += -= *= /= %=`	dx verso sx	assegnamento
`,`	sx verso dx	virgola

Figura 2.15 Precedenze degli operatori incontrati finora nel testo.

2.13 Concetti fondamentali dei cicli controllati da variabili contatore

In un ciclo di iterazione controllato da un contatore sono necessari:

1. Il *nome* della variabile di controllo (contatore del ciclo)
2. Il *valore iniziale* della variabile di controllo
3. Una condizione che verifichi se la variabile di controllo ha raggiunto il suo *valore finale* (cioè se il ciclo deve continuare o meno)
4. L'*incremento* o il *decremento*, che modifica la variabile di controllo ad ogni ripetizione del ciclo.

Consideriamo il semplice programma in Figura 2.16, che visualizza i numeri da 1 a 10. La dichiarazione

```
int counter = 1;
```

dà un nome alla variabile di controllo (`counter`), la dichiara come variabile intera, le riserva uno spazio in memoria e la imposta al *valore iniziale* di 1. Le dichiarazioni che comprendono delle inizializzazioni sono a tutti gli effetti delle istruzioni eseguibili. Se una dichiarazione riserva anche spazio in memoria si chiama *definizione*.

```
1  // Fig. 2.16: fig02_16.cpp
2  // Iterazione controllata da contatore
3  #include <iostream.h>
4
5  int main()
6  {
7     int counter = 1;              // inizializzazione
8
9     while ( counter <= 10 ) {     // condizione dell'iterazione
10       cout << counter << endl;
11       ++counter;                 // incremento
12    }
13
14    return 0;
15 }
```

```
1
2
3
4
5
6
7
8
9
10
```

Figura 2.16 Iterazione controllata da un contatore.

Avremmo potuto scrivere la dichiarazione e l'inizializzazione di `counter` in questo modo:

```
int counter;
counter = 1;
```

La dichiarazione non è un'istruzione eseguibile, al contrario dell'assegnamento. Noi utilizzeremo entrambi i metodi per inizializzare le variabili.

L'istruzione

```
++counter;
```

incrementa di 1 il contatore ad ogni ripetizione del ciclo. La condizione del costrutto `while` verifica se il valore della variabile di controllo è minore o uguale a `10`: il ciclo continuerà finché questa condizione sarà vera, cioè fino al valore `10` incluso. Il ciclo terminerà quando il contatore avrà superato questo valore, cioè al valore `11`.

Possiamo abbreviare il programma di Figura 2.16 inizializzando `counter` a `0` e cambiando il costrutto `while` in

```
while ( ++counter <= 10 )
cout << counter << endl;
```

Abbiamo risparmiato un'istruzione, dato che l'incremento si trova nella condizione della `while`, e viene effettuato prima della verifica della condizione stessa. Inoltre ora il corpo del `while` si è ridotto a una sola istruzione, per cui non abbiamo più bisogno delle parentesi graffe intorno ad esso. Con la pratica imparerete ad abbreviare il vostro codice grazie a questi stratagemmi.

Errore comune 2.12

I numeri a virgola mobile sono rappresentati in modo approssimativo nella memoria dei computer, quindi non sono adatti a essere utilizzati come variabili di controllo: la condizione di termine del ciclo potrebbe essere verificata in modo impreciso.

Buona abitudine 2.15

Nei cicli controllati da contatore conviene utilizzare soltanto variabili di controllo intere.

Buona abitudine 2.16

Rientrate le istruzioni all'interno del corpo delle strutture di controllo.

Buona abitudine 2.17

Separate le strutture di controllo dal resto del codice con delle linee vuote: in questo modo le evidenzierete maggiormente.

Buona abitudine 2.18

Se utilizzate troppi livelli di nidificazione il codice risulterà difficile da comprendere. Come regola generale, evitate più di tre livelli di nidificazione.

Buona abitudine 2.19

Se separate le strutture di controllo dal resto del codice con delle linee vuote e rientrate le istruzioni all'interno del loro corpo, il testo dei vostri programmi avrà una strutturazione definita nelle due dimensioni, cosa che si traduce in una migliore leggibilità.

2.14 La struttura di iterazione for

La struttura di iterazione `for` gestisce l'iterazione controllata da un contatore nel modo più completo. Per farvi capire la potenza e la concisione di `for` riscriviamo il programma in Figura 2.16. Il risultato è in Figura 2.17. Vediamo come funziona questo programma.

Quando inizia l'esecuzione dell'istruzione `for`, la variabile `counter` viene dichiarata e inizializzata a 1. Subito dopo viene verificata la condizione del ciclo `counter <= 10`. Dato che il valore iniziale di `counter` è 1 la condizione è soddisfatta, per cui l'istruzione del corpo di `for` visualizza il valore di `counter`, cioè 1. La variabile `counter` viene quindi incrementata in `counter++`, dopo di che il ciclo riparte da una nuova verifica della condizione. Adesso il contatore vale 2, cosa che soddisfa ancora la condizione, per cui il corpo di `for` viene eseguito di nuovo. Il processo continua finché `counter` non raggiunge il valore 11: a quel punto la condizione non è più soddisfatta e il ciclo termina. Il programma passa a eseguire l'istruzione che si trova dopo il ciclo, cioè la `return`.

La Figura 2.18 analizza il costrutto `for` di Figura 2.17. Come potete vedere, il costrutto `for` gestisce tutti gli aspetti di un ciclo controllato da contatore. Anche `for` ha un corpo di una sola istruzione, per cui se abbiamo bisogno di includerne più di una, dobbiamo racchiudere l'insieme di istruzioni tra parentesi graffe.

Fate attenzione alla condizione: infatti questa viene soddisfatta anche nel caso `counter == 10`, perché è stata scritta come `counter <= 10`. Se avessimo scritto `counter < 10` il ciclo sarebbe eseguito soltanto 9 volte. In questi casi è facile commettere un tipico errore logico che gli anglosassoni chiamano off-by-one, ovvero fuori di 1: il risultato, infatti, non è quello desiderato con lo scarto di 1 in più o in meno.

```
1  // Fig. 2.17: fig02_17.cpp
2  // Iterazione controllata da contatore con un costrutto for
3  #include <iostream.h>
4
5  int main()
6  {
7     // Inizializzazione, condizione dell'iterazione e incremento
8     // sono tutti nell'intestazione del costrutto for
9
10    for ( int counter = 1; counter <= 10; counter++ )
11       cout << counter << endl;
12
13    return 0;
14 }
```

Figura 2.17 Ciclo for per un'iterazione controllata da contatore.

Errore comune 2.13

Se utilizzate un operatore relazionale o un valore finale sbagliato, nei costrutti `while` *e* `for`*, potete commettere degli errori logici del tipo off-by-one, che comportano risultati errati con lo scarto di 1.*

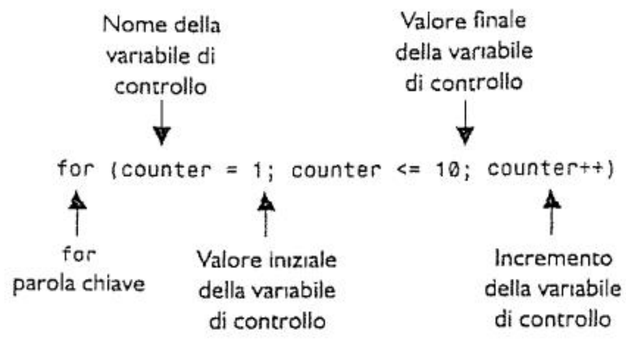


Figura 2.18 Componenti tipici dell'intestazione di un costrutto for.

Buona abitudine 2.20

Ponete il valore finale all'interno della condizione di `while` *o* `for` *e utilizzate l'operatore relazionale* `<=` *· eviterete in questo modo gli errori del tipo off-by-one. Per esempio, in un ciclo che visualizza i valori da 1 a 10, la condizione di continuazione del ciclo dovrebbe essere* `counter <= 10` *e non* `counter < 10`*, e neppure* `counter < 11`*, anche se non è errata. Ci sono molti programmatori che preferiscono inizializzare i contatori sempre a zero: un programmatore di questo genere inizializzerà* `counter` *a 0 e scriverà la condizione nella forma* `counter < 10`*.*

Il formato generico di `for` è:

```
for ( espressione1; espressione2; espressione3 )
  istruzione
```

dove *espressione1* inizializza la variabile di controllo del ciclo, *espressione2* è la condizione di continuazione del ciclo e *espressione3* incrementa la variabile di controllo. Nella maggior parte dei casi un ciclo `for` può essere tradotto facilmente in un ciclo `while` equivalente, in questo modo:

```
espressione1;
while ( espressione2 ) {
  istruzione
  espressione3;
}
```

Questa regola ha una nobile eccezione, che verrà discussa nella Sezione 2.18.

Se *espressione1* nell'intestazione di `for` definisce la variabile di controllo, cioè ne indica nome e tipo, questa non potrà essere utilizzata fuori del corpo della `for`: la variabile di controllo è nota soltanto al ciclo `for`. Questa restrizione d'uso di un nome di variabile è nota come *scope*, o *visibilità*. La visibilità di una variabile definisce in quali parti del programma può essere utilizzata. Approfondiremo l'argomento nel Capitolo 3.

Errore comune 2.14

Se definite e inizializzate la variabile di controllo di un ciclo `for` *nell'intestazione di* `for`*, non potrete utilizzare tale variabile oltre i confini del ciclo, altrimenti commetterete un errore di sintassi.*

Obiettivo portabilità 2.1

Nello standard del C++, lo scope della variabili di controllo che sono dichiarate nella sezione di inizializzazione di un costrutto `for` *è diverso da quello dei compilatori C++ meno recenti. È possibile che il codice creato su questi compilatori non funzioni correttamente su quelli più recenti, che si adeguano ai nuovi standard. Per evitare questi problemi potete adottare due strategie. La prima: definite le variabili di controllo con un nome diverso in ogni* `for`*. Se la prima non vi piace utilizzate la seconda: utilizzate pure lo stesso nome per la variabile di controllo di tutti i costrutti* `for`*, ma definitela prima di tutti i cicli e fuori di essi.*

A volte *espressione1* e *espressione3* sono liste di espressioni separate da virgole. L'operatore virgola fa sì che le espressioni nella lista siano elaborate da sinistra a destra. L'operatore virgola ha la precedenza più bassa fra tutti gli operatori del C++. Il valore e il tipo di una lista di espressioni separate da virgole coincide con il valore e il tipo dell'ultima espressione della lista. L'operatore virgola viene utilizzato spesso nei costrutti `for`, principalmente per elencare una serie di inizializzazioni o di incrementi/decrementi. Questo si verifica in particolare quando un ciclo `for` deve incrementare/decrementare più contatori.

Buona abitudine 2.21

Nelle sezioni di inizializzazione e incremento/decremento di un ciclo `for` *conviene porre soltanto espressioni che contengano le variabili di controllo. Manipolate le altre variabili prima del ciclo, se ciò è necessario una volta sola come nelle istruzioni di inizializzazione, oppure nel corpo del ciclo, se ciò è necessario a ogni ripetizione del ciclo come negli incrementi/decrementi.*

Le tre espressioni nell'intestazione della struttura `for` sono tutte opzionali. Se omettete *espressione2*, il compilatore C++ presumerà che la condizione sia sempre vera, entrando dunque in un ciclo infinito. Potete omettere *espressione1* se l'inizializzazione della variabile di controllo si trova da qualche altra parte nel programma, e potete omettere *espressione3* se l'incremento si trova nel corpo di `for` o se non c'è bisogno di alcun incremento. L'espressione di incremento dell'intestazione di `for` può essere sostituita da un'istruzione alla fine del suo corpo. Di conseguenza le espressioni

```
counter = counter + 1
counter += 1
++counter
counter++
```

per la parte incrementale di `for` sono tutte equivalenti. Molti programmatori preferiscono la forma `counter++`, perché l'incremento viene calcolato dopo l'esecuzione delle istruzioni del ciclo. La forma postincrementale quindi sembra più naturale. Tuttavia la variabile da incrementare non appare in alcuna espressione, per cui il preincremento e il postincremento sono perfettamente equivalenti. I due punti e virgola presenti nell'intestazione della struttura `for` sono obbligatori.

Errore comune 2.15

Se scrivete delle virgole al posto dei due punti e virgola dell'intestazione di `for` *commettete un errore di sintassi.*

Errore comune 2.16

Se scrivete un punto e virgola subito dopo la parentesi tonda chiusa dell'intestazione di `for`, *il corpo della struttura di controllo risulterà essere un'istruzione vuota. Fate attenzione, perché si tratta di un tipico errore logico.*

Ingegneria del software 2.9

Se scrivete un punto e virgola dopo l'intestazione di `for` *potete creare un cosiddetto* ciclo di ritardo. *L'unico scopo di un costrutto* `for` *di questo genere è essenzialmente perdere tempo in un conteggio inutile: tenetelo presente se volete rallentare l'esecuzione di un programma, per esempio se l'output scorre talmente rapidamente sullo schermo che non riuscite a leggerlo agevolmente.*

Le tre espressioni nell'intestazione di una `for` possono contenere espressioni aritmetiche. Per esempio supponiamo che `x = 2` e `y = 10`. Se `x` e `y` non sono modificati dal corpo del ciclo, l'istruzione

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

è equivalente all'istruzione

```
for ( int j = 2; j <= 80; j += 5 )
```

L'incremento di una struttura `for` può anche essere negativo: il ciclo può dunque iniziare da un valore massimo del contatore ed effettuare il conteggio alla rovescia.

Se la condizione di continuazione del ciclo è falsa già alla prima iterazione, il corpo di `for` non viene eseguito affatto.

La variabile di controllo viene spesso visualizzata o utilizzata nei calcoli presenti nel corpo di `for`, ma ciò non è indispensabile. Anzi è piuttosto comune non menzionare mai il contatore nel corpo di `for`, perché il suo unico scopo è quello di controllare il numero di iterazioni del ciclo.

Buona abitudine 2.22

Anche se è possibile modificare il valore della variabile di controllo all'interno di un ciclo `for`, *è bene evitarlo per non commettere errori logici difficili da scoprire.*

Il diagramma di flusso di `for` è molto simile a quello di `while`. In Figura 2.19 mostriamo il diagramma di questa istruzione:

```
for ( int counter = 1; counter <= 10; counter++ )
   cout << counter << endl;
```

Come risulta chiaro dal diagramma, l'inizializzazione si verifica una volta soltanto, mentre la fase di incremento viene attraversata dopo ogni esecuzione del corpo. Anche in questo caso nel diagramma troviamo solo rettangoli e rombi, oltre ai cerchietti e alle frecce. Ricordiamo brevemente che i rettangoli corrispondono alle azioni e i rombi alle decisioni.

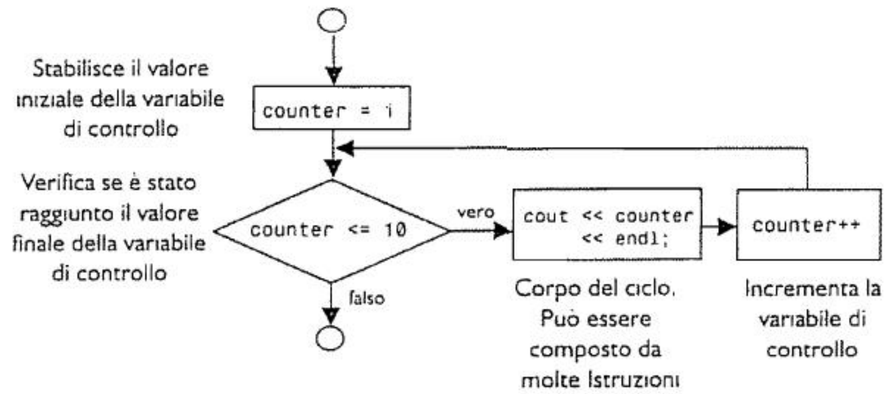


Figura 2.19 Diagramma di flusso di una tipica struttura di iterazione for.

2.15 Esempi di costrutti for

Negli esempi che seguono illustriamo come far variare il contatore di un costrutto `for` a seconda del caso da implementare. Per ogni caso scriveremo l'intestazione di `for` corrispondente. Fate attenzione agli operatori relazionali che utilizziamo.

Errore comune 2.17

Se utilizzate operatori relazionali inappropriati nella condizione di un ciclo che effettua un conteggio alla rovescia (per esempio scrivendo `i <= 1` *in un ciclo che conta fino a 1) commettete un errore logico.*

a) Far variare il contatore da `1` a `100` in incrementi di `1`.
```
for ( int i = 1; i <= 100; i++ )
```
b) Far variare il contatore da `100` a `1` in incrementi di `-1`, cioè decrementi di `1`.
```
for ( int i = 100; i >= 1; i-- )
```
c) Far variare il contatore da `7` a `77` in passi da `7`.
```
for ( int i = 7; i <= 77; i += 7 )
```
d) Far variare il contatore da `20` a `2` in passi da `-2`.
```
for ( int i = 20; i >= 2; i -= 2 )
```
e) Far assumere al contatore la serie di valori: `2, 5, 8, 11, 14, 17, 20`.
```
for ( int j = 2; j <= 20; j += 3 )
```
f) Far assumere al contatore la serie di valori: `99, 88, 77, 66, 55, 44, 33, 22, 11, 0`.
```
for ( int j = 99; j >= 0; j -= 11 )
```

Gli ultimi due esempi sono semplici applicazioni di `for`. Il programma in Figura 2.20 utilizza `for` per sommare tutti gli interi pari compresi tra `2` e `100`.

In questo caso avremmo potuto includere tutto il corpo di `for` nella sua intestazione grazie all'operatore virgola:

```
for ( int number = 2;      // inizializzazione
      number <= 100;       // condizione per la continuazione del ciclo
      sum += number, number += 2) // totale e incremento
```

Avremmo anche potuto includere in `for` anche l'inizializzazione `sum = 0`.

Buona abitudine 2.23

È possibile affollare l'intestazione di `for` *con istruzioni che andrebbero fuori e dentro il corpo del ciclo, ma se questo porta a programmi illeggibili è bene evitarlo.*

Buona abitudine 2.24

Dove possibile, limitate le intestazioni delle strutture di controllo a una sola linea.

```
1   // Fig. 2.20: fig02_20.cpp
2   // Sommatoria con for
3   #include <iostream.h>
4
5   int main()
6   {
7      int sum = 0;
8
9      for ( int number = 2; number <= 100; number += 2 )
10        sum += number;
11
12     cout << "Sum is " << sum << endl;
13
14     return 0;
15  }
```

```
Sum is 2550
```

Figura 2.20 Una sommatoria eseguita con for.

Nel prossimo esempio calcoleremo un interesse composto con `for`. Ecco la definizione del nostro problema:

Un individuo investe $1000.00 in un conto corrente con un interesse del 5%. Se l'interesse viene lasciato in deposito sul conto, calcolate la somma che si trova sul conto alla fine di ogni anno nel prossimo decennio. Per determinare questo valore utilizzate la formula:

$a = p\ (1 + r)^n$

dove
p è il capitale investito,
r è il tasso di interesse annuale,
n è il numero di anni
a è la somma in deposito alla fine dell'anno n-esimo.

Per risolvere questo problema scriviamo un ciclo che effettua il calcolo per ognuno dei 10 anni in cui la somma rimane in deposito. La soluzione è in Figura 2.21.

Il ciclo di 10 iterazioni viene effettuato da un costrutto `for` che fa variare una variabile di controllo da 1 a 10 in incrementi di 1. In C++ non ha un operatore esponente predefinito, per cui dobbiamo ricorrere alla funzione di libreria `pow`: la funzione `pow(x, y)` calcola `x` alla `y`-esima potenza. La funzione `pow` prende due argomenti di tipo `double` e restituisce un valore pure di tipo `double`. Il tipo `double` è simile a `float`, perché è anch'esso un tipo di dato a virgola mobile, ma può memorizzare valori molto più grandi e

con una precisione di gran lunga maggiore di `float`. In C++ le costanti in virgola mobile come `1000.0` e `.05` sono trattate automaticamente come valori `double`.

```
1   // Fig. 2.21: fig02_21.cpp
2   // Calcolo dell'interesse composto
3   #include <iostream.h>
4   #include <iomanip.h>
5   #include <math.h>
6
7   int main()
8   {
9      double amount,           // somma in deposito
10            principal = 1000.0,  // capitale iniziale
11            rate = .05;          // tasso di interesse
12
13     cout << "Year" << setw( 21 )
14          << "Amount on deposit" << endl;
15
16     for ( int year = 1; year <= 10; year++ ) {
17        amount = principal * pow( 1.0 + rate, year );
18        cout << setw( 4 ) << year
19             << setiosflags( ios::fixed | ios::showpoint )
20             << setw( 21 ) << setprecision( 2 )
21             << amount << endl;
22     }
23
24     return 0;
25  }
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.62
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Figura 2.21 Calcolo dell'interesse composto con for.

Non possiamo compilare questo programma senza includere il file `math.h`. Abbiamo detto che la funzione `pow` prende due argomenti di tipo `double`: ma `year` è un intero. Il file `math.h` include le informazioni necessarie al compilatore per convertire il valore di `year` in una rappresentazione `double` temporanea. A conversione effettuata il compilatore può generare il codice per chiamare la funzione `pow`. Queste informazioni sono contenute nel prototipo di funzione di `pow`. Vedremo cosa sono i prototipi di funzione nel Capitolo 3. Avremo modo anche di riprendere il discorso su `pow` e sulle altre funzioni matematiche.

Errore comune 2.18

Se dimenticate di includere il file `math.h` *in un programma che utilizza funzioni matematica di libreria commettete un errore di sintassi.*

Come notate, abbiamo dichiarato come `double` le variabili `amount`, `principal` e `rate`. Il motivo è che avremo a che fare con parti decimali di dollaro, per cui abbiamo bisogno di un tipo di dato che preveda i decimali. In realtà se questo risolve il problema, da un altro punto di vista lo complica. Vediamo un po' cosa può andare storto quando si rappresentano i capitali con i tipi `float` o `double`; supponiamo di visualizzarli sempre con `setprecision(2)`. Supponiamo di avere due somme di denaro `double`, come per esempio 14.234 (visualizzata come 14.23) e 18.673 (visualizzata come 18.67). Quando i due valori vengono sommati producono la somma interna 32.907, che viene visualizzata come 32.91. Questa operazione dunque può apparire così:

```
  14.23
+ 18.67
-------
  32.91
```

ma ora effettuate l'operazione manualmente: avrete un altro risultato, cioè 32.90! La controprova non torna.

Buona abitudine 2.25

Evitate di effettuare i calcoli finanziari con variabili di tipo `float` *o* `double`. *L'imprecisione di questi tipi di dato si riflette in risultati imprecisi o scorretti. Negli esercizi vedremo come utilizzare gli interi nei calcoli finanziari. Aggiungiamo anche, però, che le moderne librerie di classi del C++ si attrezzano ogni giorno di più per effettuare calcoli finanziari precisi.*

L'istruzione di output

```
cout << setw( 4 ) << year
     << setiosflags( ios::fixed | ios::showpoint )
     << setw( 21 ) << setprecision( 2 )
     << amount << endl;
```

visualizza il valore di `year` e `amount`, con la formattazione indicata dai manipolatori di stream parametrizzati `setw`, `setiosflags` e `setprecision`. La chiamata a `setw(4)` specifica che il valore successivo in output verrà visualizzato con una larghezza di campo di 4, e cioè che verranno visualizzate almeno 4 posizioni. Se il valore da inviare in output ha un'ampiezza minore di 4 caratteri, per default verrà giustificato a destra nel campo. Se invece i caratteri da visualizzare sono più di 4, la larghezza del campo deve essere estesa, per consentire la visualizzazione dell'intero valore. Scrivete `setiosflags(ios::left)` per indicare una giustificazione a sinistra.

Il resto della formattazione dell'istruzione precedente indica che si deve visualizzare `amount` come valore a virgola fissa con il punto decimale (specificato in `setiosflags( ios::fixed | ios::showpoint )`), con giustificazione a destra in un campo di 21 posizioni (specificato in `setw( 21 )`) e con due cifre di precisione a destra del punto decimale (specificato in `setprecision( 2 )`). Nel Capitolo 11 torneremo sulle funzionalità di formattazione del C++ per l'input e l'output.

Il calcolo `1.0 + rate`, argomento di `pow`, è contenuto nel corpo di `for`. Tuttavia il calcolo produce lo stesso risultato in tutta l'iterazione, per cui ripeterlo continuamente lo consideriamo uno spreco di tempo e di elaborazione.

Obiettivo efficienza 2.5

Evitate di includere nei cicli espressioni che producono sempre lo stesso valore. Anche se vi capita di farlo, i moderni compilatori provvedono automaticamente a spostarle fuori del ciclo quando generano il codice in linguaggio macchina.

Obiettivo efficienza 2.6

Diversi compilatori hanno delle funzionalità di ottimizzazione che migliorano automaticamente il codice che voi scrivete, ma resta sempre preferibile scrivere del buon codice fin dall'inizio.

Provate a divertirvi con l'esercizio 2.65, a fine capitolo. Farete conoscenza con le meraviglie dell'interesse composto.

2.16 La struttura di selezione switch

Finora abbiamo parlato della struttura di selezione singola `if` e di selezione doppia `if/else`. In alcuni algoritmi si deve scegliere tra una serie di decisioni: a seconda del valore assunto da una variabile o da un'espressione si decide un cammino di esecuzione diverso. Questa struttura decisionale è implementata nel C++ dall'istruzione `switch`.

Un costrutto `switch` si compone di una serie di etichette `case` e in via opzionale di un'etichetta `default`. Il programma in Figura 2.22 utilizza `switch` per contare il numero di voti differenti assegnati durante un esame. Anziché in numeri, i voti sono espressi in lettere, secondo il sistema americano: A indica il voto massimo, B, C, D e F indicano voti via via più bassi.

```
1   // Fig. 2.22: fig02_22.cpp
2   // Conteggio dei voti
3   #include <iostream.h>
4
5   int main()
6   {
7      int grade,        // un voto
8          aCount = 0,   // numero di A
9          bCount = 0,   // numero di B
10         cCount = 0,   // numero di C
11         dCount = 0,   // numero di D
12         fCount = 0;   // numero di F
13
14     cout << "Enter the letter grades." << endl
15          << "Enter the EOF character to end input." << endl;
16
17     while ( ( grade = cin.get() ) != EOF ) {
18
```

Figura 2.22 Esempio di costrutto switch (continua)

```
19        switch ( grade ) {     // switch nidificato in while
20
21           case 'A': // il voto era una lettera A maiuscola
22           case 'a'   // o una a minuscola
23              ++aCount;
24              break;  // necessario per uscire da switch
25
26           case 'B':  // il voto era una lettera B maiuscola
27           case 'b':  // o una b minuscola
28              ++bCount;
29              break;
30
31           case 'C':  // il voto era una lettera C maiuscola
32           case 'c':  // o una c minuscola
33              ++cCount;
34              break;
35
36           case 'D':  // il voto era una lettera D maiuscola
37           case 'd':  // o una d minuscola
38              ++dCount;
39              break;
40
41           case 'F':  // il voto era una lettera F maiuscola
42           case 'f':  // o una f minuscola
43              ++fCount;
44              break;
45
46           case '\n': // ignora i caratteri di nuova linea,
47           case '\t': // le tabulazioni,
48           case ' ':  // e gli spazi in input
49              break;
50
51           default:   // intercetta tutti gli altri caratteri
52              cout << "Incorrect letter grade entered."
53                   << " Enter a new grade." << endl;
54              break;  // opzionale
55        }
56     }
57
58     cout << "\n\nTotals for each letter grade are:"
59          << "\nA: " << aCount
60          << "\nB: " << bCount
61          << "\nC: " << cCount
62          << "\nD: " << dCount
63          << "\nF: " << fCount << endl;
64
65     return 0;
66  }
```

Figura 2.22 Esempio di costrutto switch (continua)


```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Figura 2.22 Esempio di costrutto switch.

Durante l'esecuzione il programma chiede all'utente di immettere i voti di una determinata classe. Nell'intestazione del `while`

```
while ( ( grade = cin.get() ) != EOF)
```

viene eseguito per primo l'assegnamento fra parentesi (`grade = cin.get()`). La funzione `cin.get()` legge un carattere dalla tastiera e lo memorizza nella variabile intera `grade`. La notazione `cin.get()` verrà spiegata nel Capitolo 6. Normalmente i caratteri sono memorizzati in variabili di tipo `char`. Tuttavia esiste un'importante caratteristica del C++, che prevede che essi possano essere memorizzati in qualsiasi tipo di dato intero: all'interno del computer infatti i caratteri sono già rappresentati come interi di 1 byte. Di conseguenza, a seconda dei casi, abbiamo la libertà di trattare un carattere come numero o come carattere alfanumerico. Per esempio, l'istruzione

```
cout << "The character (" << 'a' << ") has the value "
     << static_cast< int > ( 'a' ) << endl;
```

visualizza il carattere `a` e il valore intero corrispondente:

```
The character (a) has the value 97
```

l'intero `97` corrisponde alla rappresentazione numerica di `a` all'interno del computer. Molti computer oggi utilizzano il set di caratteri ASCII (American Standard Code for Information Interchange), dove 97 corrisponde alla lettera minuscola `'a'`. Nelle appendici trovate l'elenco dei caratteri ASCII, con i valori decimali corrispondenti.

Un'istruzione di assegnamento, nel complesso, assume il valore assegnato alla variabile a sinistra del segno =. Perciò il valore dell'assegnamento `grade = cin.get()` è uguale a quello restituito da `cin.get()` e assegnato alla variabile `grade`.

Questa proprietà degli operatori di assegnamento torna utile se si deve inizializzare una serie di variabili allo stesso valore. Per esempio,

```
a = b = c = 0;
```

calcola prima l'assegnamento `c = 0`, perché l'operatore = associa da destra a sinistra. A `b` viene assegnato il valore dell'assegnamento `c = 0`, cioè 0. Infine ad `a` viene assegnato il valore dell'assegnamento `b = (c = 0)`, che è ancora zero. Nel programma il valore dell'assegnamento `grade = cin.get()` viene confrontato con il valore `EOF`, simbolo che sta per l'inglese end-of-file, ovvero fine del file. Potete vedere `EOF` come una sorta di valore sentinella: il suo valore è generalmente uguale a −1. La fine del file rappresenta la fine dei dati in input, e su ogni sistema corrisponde a una data combinazione di caratteri. `EOF` è una costante intera simbolica definita nel file di intestazione `<iostream.h>` Se il valore assegnato a `grade` è uguale a `EOF`, il programma termina. In questo caso la scelta di rappresentare i caratteri con degli interi è intelligente perché `EOF` normalmente ha il valore intero −1.

Obiettivo portabilità 2.2

La combinazione di caratteri che indica la fine del file dipende dal sistema.

Obiettivo portabilità 2.3

Verificate se avete raggiunto la fine del file con la costante `EOF` *anziché con il valore* −1, *per scrivere programmi più portabili. Su alcuni sistemi, infatti,* `EOF` *potrebbe avere un valore diverso: lo standard ANSI stabilisce semplicemente che* `EOF` *è un valore intero negativo, non necessariamente uguale a* −1.

Su molti sistemi, inclusi i sistemi UNIX, la fine del file si segnala digitando la sequenza

<ctrl-d>

su una linea singola. Questa notazione indica che bisogna digitare la lettera `d` tenendo premuto il tasto `ctrl`. Su altri sistemi, come VAX VMS di Digital Equipment Corporation o MS-DOS di Microsoft, la fine del file si segnala digitando

<ctrl-z>

Ora torniamo al nostro programma. L'utente deve digitare i voti alla tastiera. Ogni volta che viene premuto il tasto Invio la funzione `cin.get()` legge i caratteri digitati, uno alla volta. Se la funzione non trova il carattere `EOF`, si entra nel costrutto `switch`. Tra parentesi troviamo la variabile `grade`: essa costituisce la cosiddetta espressione di controllo. Il valore di `grade` sarà messo a confronto con ciascuna delle etichette `case`. Ora supponiamo che l'utente abbia digitato la lettera `C`. Il programma mette a confronto la lettera `C` con ogni etichetta `case`, finché non trova un uguaglianza (`case 'C':`): una volta trovata l'uguaglianza, vengono eseguite le istruzioni che seguono la clausola `case` corrispondente. Nel nostro caso `cCount` verrà incrementato di 1, e il costrutto `switch` terminerà immediatamente dopo con l'istruzione `break`. A differenza delle altre strutture di controllo, qui non abbiamo bisogno di racchiudere più istruzioni di una `case` tra parentesi graffe.

L'istruzione `break` fa saltare il flusso del programma alla prima istruzione che segue il costrutto `switch`. L'uso di `break` è necessario, perché altrimenti i `case` andrebbero tutti in esecuzione uno dopo l'altro: in mancanza di un'istruzione `break`, una volta trovato un `case` che soddisfa l'uguaglianza, sono eseguite tutte le istruzioni del costrutto `switch` fino

alla fine, comprese quindi quelle relative a tutti i `case` successivi. Questa caratteristica esiste per consentire l'esecuzione delle stesse operazioni per una serie di `case`, come nel programma in Figura 2.22. Se non si trova alcun `case` uguale all'espressione di controllo, viene eseguita la clausola `default`, che visualizza un messaggio di errore.

Ogni clausola `case` può prevedere una o diverse operazioni. In questo `switch` è differente da tutte le altre strutture di controllo, perché, l'abbiamo già detto, non sono necessarie le parentesi graffe per delimitarle. Il diagramma di flusso della struttura di selezione multipla `switch` (con tutti i `break` dopo le clausole `case`) è in Figura 2.23.

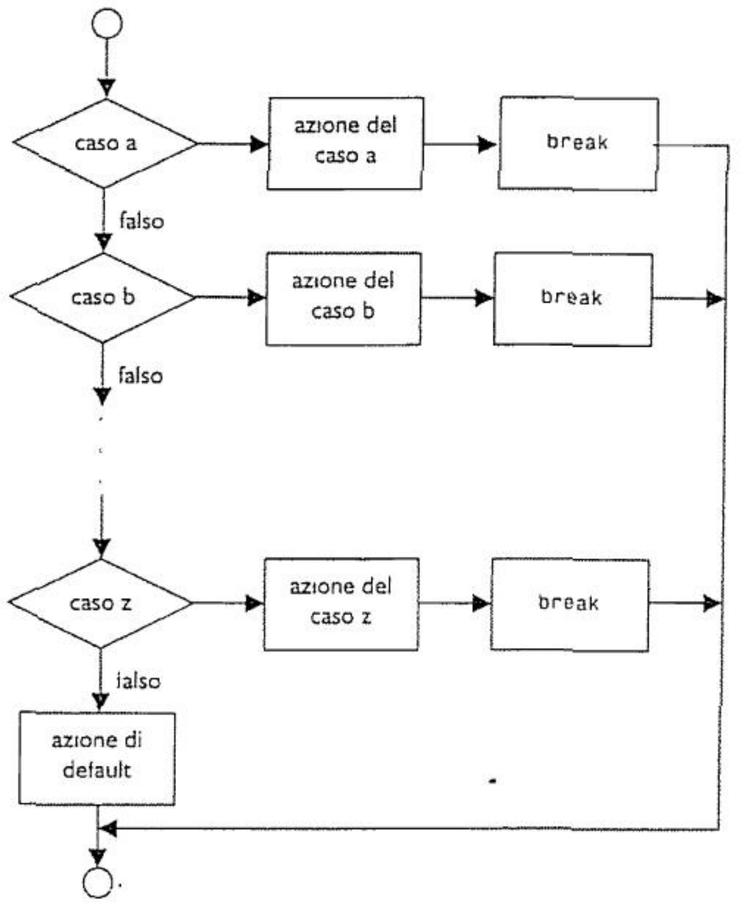


Figura 2.23 Uso di break in un costrutto switch.

Il diagramma evidenzia il salto del flusso all'uscita dell'intero costrutto `switch` dopo ogni `break`. Anche in questo caso il diagramma contiene solo rettangoli e rombi, a parte i cerchietti e le frecce. Ricordiamo che i rettangoli rappresentano le azioni e i rombi le decisioni. È raro trovare strutture `switch` nidificate.

Errore comune 2.19

Se dimenticate di scrivere un'istruzione `break` *in un costrutto* `switch`, *dove è necessario, commettete un errore logico.*

Errore comune 2.20

Se omettete lo spazio tra la parola riservata `case` *e il valore intero verificato nel costrutto* `switch` *commetterete un errore logico. Per esempio se scrivete* `case3:` *anziché* `case 3:` *creerete semplicemente un'etichetta non utilizzata denominata* `case3` *(lo vedremo nel Capitolo 7 del volume Tecniche Avanzate), nulla a che vedere con una clausola di* `switch`. *Il problema è che il costrutto* `switch` *non effettuerà l'operazione prevista nel caso che l'espressione di controllo abbia il valore* `3`.

Buona abitudine 2.26

Conviene prevedere sempre una clausola `default` *in un* `switch`. *In sua assenza, i casi non previsti esplicitamente dalle clausole* `case` *verranno ignorati. Includendo una clausola* `default` *prenderete in considerazione anche condizioni eccezionali. Tuttavia conveniamo che ci sono situazioni in cui non c'è alcun bisogno di una clausola* `default`.

Buona abitudine 2.27

Anche se le clausole `case` *e* `default` *possono comparire in qualsiasi ordine in un costrutto* `switch`, *è considerata una buona abitudine porre come ultima la clausola* `default`.

Buona abitudine 2.28

Se ponete la clausola `default` *come ultima in un costrutto* `switch`, *non è necessario includere un'istruzione* `break`. *Alcuni programmatori tuttavia la scrivono ugualmente per chiarezza e simmetria rispetto agli altri casi.*

Nell'istruzione `switch` di Figura 2.22, le linee dalla 46 alla 49

```
case '\n':
case '\t':
case ' ':
   break;
```

fanno sì che il programma salti le spaziature, i caratteri di nuova linea e le tabulazioni. Leggere i caratteri uno alla volta può anche generare un problema. Perché il computer possa leggerli occorre premere Invio dopo aver digitato ciascun carattere: in input sarà presente però anche un carattere di nuova linea, oltre ai caratteri che si vogliono leggere. In genere questo carattere deve essere trattato in un modo speciale, altrimenti il programma non funzionerà correttamente. Adesso includiamo gli altri casi precedenti in `switch`, in modo da evitare il messaggio di errore della clausola `default` per ogni spaziatura, tabulazione o carattere di nuova linea.

Errore comune 2.21

Prendete in considerazione i caratteri di spaziatura presenti nell'input, o potreste commettere errori logici.

Abbiamo elencato di seguito diverse clausole (come `case 'D':` e `case 'd'` in Figura 2.22): ciò significa semplicemente che esse prevedono l'esecuzione dello stesso insieme di operazioni.

Ricordate che potete utilizzare `switch` soltanto per verificare espressioni intere costanti, cioè qualsiasi combinazione di costanti carattere e intere che abbia un valore intero costante. Una costante carattere si rappresenta con il carattere specifico tra una coppia di apici singoli, come per esempio `'A'` Una costante intera è semplicemente il valore intero.

Quando parleremo della programmazione orientata agli oggetti, presenteremo un'implementazione più elegante della logica di `switch`. Grazie al polimorfismo riusciremo a scrivere programmi più leggibili, più concisi e più semplici da mantenere ed espandere rispetto ai programmi che fanno uso di `switch`.

I linguaggi portabili come il C++ devono possedere tipi di dato di dimensioni flessibili. Applicazioni diverse possono aver bisogno di interi di dimensioni diverse. Il C++ rende disponibili diversi tipi di dato per la rappresentazione di numeri interi. La gamma di valori rappresentabile con i diversi tipi dipende dall'hardware del proprio computer. Oltre ai tipi `int` e `char`, il C++ prevede i tipi `short` (abbreviazione di `short int`, intero piccolo) e `long` (abbreviazione di `long int`, intero grande). L'intervallo minimo di un intero `short` va da –32767 a +32767. Per la maggior parte dei calcoli interi, i numeri `long` sono più che sufficienti. L'intervallo minimo di un `long` va da –2147483647 a +2147483647. Sulla maggior parte dei computer il tipo `int` equivale a `short` o `long`. La gamma di valori per un `int` è definita non minore di quella di uno `short` e non maggiore di quella di un `long`. Il tipo `char` può essere utilizzato per rappresentare qualsiasi carattere del set disponibili sul computer specifico. Il tipo `char`, come abbiamo detto, può rappresentare anche piccoli numeri interi.

Obiettivo portabilità 2.4

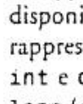
Gli `int` *possono avere dimensione diversa su sistemi diversi, per cui conviene utilizzare i numeri* `long` *nei calcoli che probabilmente daranno risultati al di fuori dell'intervallo –32.767 / +32.767.*

Obiettivo efficienza 2.7

Se siete alle prese con un problema in cui le prestazioni sono di cruciale importanza, o l'utilizzo di memoria deve essere contenuto il più possibile, utilizzate interi di dimensione più piccola (`char`, `short` *o* `int`*).*

Obiettivo efficienza 2.8

Può capitare che l'utilizzo di interi più piccoli possa produrre programmi meno veloci, perché le istruzioni in linguaggio macchina che li manipolano a volta non sono altrettanto efficienti di quelle che operano sugli interi più grandi (per es. devono prima subire un'operazione detta estensione del segno).



Errore comune 2.22

Se in un costrutto `switch` *scrivete più clausole* `case` *identiche commettete un errore di sintassi.*

2.17 La struttura iterativa do/while

Il costrutto `do/while` è simile al costrutto `while`. In quest'ultimo la condizione di continuazione del ciclo si trova all'inizio, prima dell'esecuzione delle istruzioni. Nel costrutto `do/while` invece la condizione si trova dopo il corpo del ciclo, per cui il ciclo viene

eseguito comunque, almeno una volta. Se il corpo di un `do/while` contiene una sola istruzione, non è necessario racchiuderla tra parentesi graffe. Tuttavia alcuni preferiscono scriverle comunque, per evitare di confondersi tra i due costrutti `while` e `do/while`. Per esempio,

```
while ( condition )
```

appare come l'intestazione di un costrutto `while`. Un `do/while` senza parentesi graffe ha questo aspetto:

```
do
   statement
while ( condition );
```

Come vedete, è possibile confondersi. L'ultima linea può essere confusa come un costrutto `while` con un'istruzione vuota. È per questo che normalmente le parentesi sono presenti anche se non ce ne sarebbe bisogno:

```
do {
   statement
} while ( condition );
```

Buona abitudine 2.29

Alcuni programmatori scrivono ugualmente le parentesi graffe per delimitare il corpo di un costrutto `do/while`*, anche se non sono necessarie. Ciò è utile per evitare di confondere un costrutto* `do/while` *che contiene una sola istruzione con un costrutto* `while`*.*

Errore comune 2.23

Se le condizioni contenute nei costrutti `while`*,* `for` *o* `do/while` *non diventano mai false, si avranno dei cicli infiniti. Per evitarli, assicuratevi che il valore della variabile condizionale cambi da qualche parte, nell'intestazione o nel corpo del costrutto, in modo che a un certo punto la condizione possa diventare falsa.*

Il programma in Figura 2.24 utilizza una struttura di ripetizione `do/while` per visualizzare i numeri da 1 a 10. Notate che il contatore `counter` viene preincrementato all'interno del test per la continuazione del ciclo. Notare anche l'uso delle parentesi graffe per racchiudere l'unica istruzione del costrutto.

```
1  // Fig. 2.24: fig02_24.cpp
2  // La struttura di iterazione do/while
3  #include <iostream.h>
4
5  int main()
6  {
7     int counter = 1;
8
9     do {
10       cout << counter << "  ";
11    } while ( ++counter <= 10 );
12
13    cout << endl;
14
15    return 0;
16 }
```

```
    1  2  3  4  5  6  7  8  9  10
```

Figura 2.24 Esempio di costrutto do/while (continua)

Figura 2.24 Esempio di costrutto do/while.

Il diagramma di flusso del `do/while` è in Figura 2.25. Come è evidente dal diagramma, la condizione non viene presa in considerazione finché il corpo del ciclo non viene eseguito completamente almeno una volta. Anche qui notiamo che gli unici simboli presenti nel diagramma sono rettangoli e rombi, a parte i cerchietti e le frecce. Ricordiamo che i rettangoli rappresentano le azioni e i rombi le decisioni.

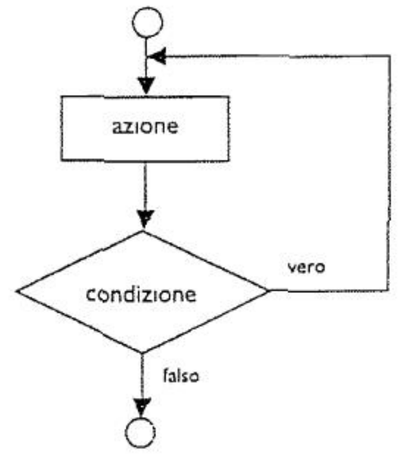


Figura 2.25 Diagramma di flusso della struttura di iterazione do/while

2.18 Le istruzioni break e continue

Il compito delle istruzioni `break` e `continue` è alterare il flusso di controllo del programma. L'istruzione `break` eseguita all'interno dei costrutti `while`, `do/while` o `switch` causa l'uscita immediata dal costrutto. L'esecuzione continua dall'istruzione che si trova immediatamente dopo il costrutto. Comunemente si usa `break` per uscire prematuramente da un ciclo o per saltare le istruzioni restanti di un costrutto `switch` (come in Figura 2.22). In *Figura 2.26 avete un esempio di* `break` *utilizzato in un ciclo* `for`*. Quando* `if` *verifica che* `x` è diventato uguale a `5`, viene eseguita l'istruzione `break`. Questo causa il termine del ciclo `for`, facendo saltare il programma alla `cout` successiva. Il ciclo quindi viene eseguito completamente soltanto quattro volte.

Come vedete la variabile di controllo `x` è stata definita fuori del ciclo `for`: questo perché la nostra intenzione è quella di utilizzarla anche dopo il ciclo.

```
1   // Fig. 2.26: fig02_26.cpp
2   // Uso di break in un ciclo for
3   #include <iostream.h>
4
5   int main()
6   {
7   // dichiaramo x qui cosi potremo utilizzarlo anche dopo il ciclo
8      int x;
9
10     for ( x = 1; x <= 10; x++ ) {
11
12        if ( x == 5 )
13           break;     // termina il ciclo solo se x == 5
14
15        cout << x << " ";
16     }
17
18     cout << "\nBroke out of loop at x of " << x << endl;
19     return 0;
20  }
```

```
1 2 3 4
Broke out of loop at x of 5
```

Figura 2.26 L'istruzione break e il costrutto for

L'istruzione `continue`, eseguita all'interno dei costrutti `while`, `for` o `do/while`, salta le istruzioni restanti del costrutto e continua il ciclo dall'iterazione successiva. Nei costrutti `while` e `do/while` la condizione di continuazione del ciclo viene valutata immediatamente dopo l'esecuzione di `continue`. In precedenza abbiamo detto che il costrutto `while` può sostituire quasi sempre un costrutto `for`. Il quasi si riferiva a un'unica eccezione, quando l'espressione incrementale di un `while` si trova dopo un'istruzione `continue`. In questo caso l'incremento non viene eseguito prima che la condizione non sia nuovamente verificata: in questo caso dunque il comportamento di `for` e `while` è diverso. In Figura 2.27 usiamo `continue` in un `for` per saltare l'istruzione di output e ricominciare dall'iterazione successiva del ciclo.

```
1   // Fig. 2.27: fig02_07.cpp
2   // Uso di continue in un ciclo for
3   #include <iostream.h>
4
5   int main()
6   {
7      for ( int x = 1; x <= 10; x++ ) {
8
9         if ( x == 5 )
10           continue;  // salta il codice restante del ciclo
11                      // solo se x == 5
```

Figura 2.27 L'istruzione continue e il costrutto for (continua)


```
12
13          cout << x << " ";
14       }
15
16       cout << "\nUsed continue to skip printing the value 5"
17            << endl;
18       return 0;
19    }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Figura 2.27 L'istruzione continue e il costrutto for.

Buona abitudine 2.30

Alcuni programmatori pensano che le istruzioni `break` *e* `continue` *violino i principi della programmazione strutturata. In realtà esistono delle tecniche di programmazione strutturata che possono sempre sostituirle, per cui questi programmatori preferiscono evitarle nei loro programmi.*

Obiettivo efficienza 2.9

Utilizzate in modo appropriato, le istruzioni `break` *e* `continue` *generano un codice macchina più efficiente rispetto alle tecniche di programmazione strutturata corrispondenti.*

Ingegneria del software 2.10

C'è una tensione continua tra i due obiettivi principali della scrittura di software: scrivere codice di qualità dal punto di vista formale, e scrivere codice più efficiente. Spesso se si sceglie di privilegiare un obiettivo, l'altro ne viene influenzato negativamente.

2.19 Gli operatori logici

Finora abbiamo incontrato condizioni semplici, come `counter <= 10`, `total > 1000` e `number != sentinelValue`. Abbiamo espresso queste condizioni per mezzo degli operatori relazionali `>`, `<`, `>=` e `<=`, e degli operatori di uguaglianza `==` e `!=`. In una condizione semplice ogni decisione verifica una sola condizione. Per verificare più condizioni in un processo decisionale abbiamo effettuato i test in istruzioni separate oppure in costrutti `if` o `if/else` nidificati.

In C++ è possibile sfruttare i cosiddetti operatori logici per formare condizioni complesse a partire da più condizioni semplici. Gli operatori logici sono `&&` (AND logico), `||` (OR logico) e `!` (NOT logico o negazione logica). Daremo un esempio per ognuno di essi.

Supponiamo di voler verificare se due condizioni sono entrambe vere, per scegliere un determinato percorso all'interno del programma. In questo caso utilizziamo l'operatore logico `&&` così:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

L'istruzione `if` contiene due condizioni semplici. La condizione `gender == 1` verifica, per esempio, se una determinata persona è di sesso femminile. La condizione `age >= 65` invece verifica se è un pensionato. Viene verificata per prima la condizione semplice a sinistra dell'operatore `&&`, perché la precedenza di `==` è più alta di quella di `&&`. Se è necessario, viene verificata la condizione semplice a destra dell'operatore `&&`, perché la precedenza di `>=` è più alta di quella di `&&`: come vedremo, il lato destro di un'espressione che contiene un AND logico viene verificato solo se l'espressione sul lato sinistro risulta già vera.

L'istruzione `if` prende in considerazione l'intera espressione

```
gender == 1 && age >= 65
```

Questa condizione è vera se e solo se entrambe le condizioni semplici che la compongono sono vere. Se è così, `seniorFemales` viene incrementato di 1. Se una o entrambe le condizioni sono false il programma salta l'istruzione incrementale. Possiamo migliorare la leggibilità di questa condizione scrivendo delle parentesi ridondanti

```
( gender == 1 ) && ( age >= 65 )
```

Errore comune 2.24

Anche se la condizione `3 < x < 7` *è corretta dal punto di vista matematico, non è calcolata come immaginate in C++. La conversione di questa notazione matematica in C++ è:* `(3 < x && x < 7)`.

La tabella in Figura 2.28 riepiloga l'uso dell'operatore `&&`. In essa trovate le quattro combinazioni possibili di valori `true` e `false` per *espressione1* e *espressione2*. Le tabelle di questo tipo prendono il nome di tabelle di verità. In C++ tutte le espressioni che includono operatori relazionali, di uguaglianza e/o logici assumono valore `true` o `false`.

espressione1	espressione2	espressione1 && espressione2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 2.28 Tabella di verità dell'operatore && (AND logico).

Obiettivo portabilità 2.5

Per assicurare la compatibilità dei valori di tipo `bool` *con le versioni precedenti dello standard del C++, il valore* `true` *è rappresentato da qualsiasi valore diverso da zero, e il valore* `false` *dal valore* `0`.

Vediamo adesso come funziona l'operatore `||` (OR logico). Ci sono casi in cui vogliamo prendere in considerazione due condizioni, per verificare se una o entrambe sono vere, prima di scegliere un determinato percorso di esecuzione. In questi casi si utilizza l'operatore `||`. Osservate questo segmento di programma:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   cout << "Student grade is A" << endl;
```


Anche questa condizione contiene due condizioni semplici. La condizione semplice `semesterAverage >= 90` viene verificata per determinare se lo studente in questione merita il giudizio massimo "A" nel corso, per aver ottenuto ottimi risultati durante il semestre. La condizione semplice `finalExam >= 90` determina se lo studente merita il giudizio "A" per aver superato l'esame finale in modo brillante. L'istruzione `if` prende in considerazione entrambe le condizioni

```
semesterAverage >= 90 || finalExam >= 90
```

e premia lo studente con l'agognato giudizio "A" nel caso una o entrambe siano vere. Notate che il messaggio `"Student grade is A"` non verrà visualizzato soltanto se entrambe le condizioni risultano false. In Figura 2.29 troviamo la tabella di verità per l'operatore OR logico (`||`).

L'operatore `&&` ha precedenza più alta di `||`. Entrambi *associano da sinistra a destra*. La verifica di un'espressione che contiene gli operatori `&&` e `||` va avanti fino a quando non viene stabilito con certezza se essa è vera o falsa. A quel punto si interrompe. Per esempio, la verifica dell'espressione

```
gender == 1 && age >= 65
```

si interrompe subito se `gender` non è uguale a 1 (l'intera espressione è falsa), mentre esso continua se `gender` è 1 (l'intera espressione potrebbe essere ancora sia vera che falsa, dipende ora dalla condizione `age >= 65`).

espressione1	espressione2	espressione1 || espressione2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 2.29 Tabella di verità dell'operatore || (OR logico).

Errore comune 2.25

In espressioni che contengono l'operatore `&&`*, è possibile che una condizione debba essere vera perché un'altra (che chiameremo condizione dipendente) abbia significato. In questo caso la condizione dipendente dovrebbe essere scritta prima dell'altra, altrimenti potrebbe verificarsi un errore.*

Obiettivo efficienza 2.10

Nelle espressioni che contengono l'operatore `&&`*, se le due condizioni sono indipendenti l'una dall'altra, scrivete per prima quella che ha probabilità maggiori di essere falsa. In espressioni che contengono l'operatore* `||`*, scrivete per prima quella che ha probabilità maggiori di essere vera.*

Passiamo ora all'operatore di negazione logica, `!`, che permette di rovesciare il valore di una condizione. A differenza di `&&` e `||`, che sono operatori binari, l'operatore di negazione logica si applica a una sola condizione, ed è quindi un operatore unario. L'operatore di negazione logica si scrive prima di una condizione, quando si vuole scegliere un particolare percorso di esecuzione se tale condizione risulta falsa. Ecco un segmento di programma esemplificativo:

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade << endl;
```

Le parentesi che racchiudono la condizione `grade == sentinelValue` sono necessarie, perché la negazione logica ha precedenza più alta dell'operatore di uguaglianza. In Figura 2.30 presentiamo la tabella di verità per l'operatore di negazione logica.

espressione	!espressione
`false`	`true`
`true`	`false`

Figura 2.30 Tabella di verità dell'operatore ! (negazione logica).

Nella maggior parte dei casi, comunque, è possibile riscrivere l'espressione in modo diverso, senza utilizzare la negazione logica. Basta utilizzare gli operatori relazionali e di uguaglianza appropriati. Per esempio l'istruzione precedente può essere scritta in questo modo:

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade << endl;
```

Questa flessibilità aiuta a esprimere una condizione nel modo più naturale, o conveniente, possibile.

La Figura 2.31 illustra precedenza e associatività di tutti gli operatori che abbiamo introdotto finora. Li elenchiamo in ordine decrescente di precedenza.

Operatori	Associatività	Tipo
`()`	sx verso dx	parentesi
`++ -- + - ! static_cast<type> ()`	dx verso dx	unario
`* / %`	sx verso dx	moltiplicativo
`+ -`	sx verso dx	additivo
`<< >>`	sx verso dx	inserzione/estrazione
`< <= > >=`	sx vero dx	relazionale
`== !=`	sx vero dx	di uguaglianza
`&&`	sx vero dx	AND logico
`||`	sx vero dx	OR logico
`?:`	dx verso dx	condizionale
`= += -= *= /= %=`	dx verso dx	assegnamento
`,`	sx vero dx	virgola

Figura 2.31 Precedenza e associatività degli operatori.

2.20 Un errore tipico: confondere l'operatore di uguaglianza == con l'operatore di assegnamento =

È un errore talmente comune, anche fra i programmatori esperti, che abbiamo deciso di dedicargli una sezione apposita. È facile scambiare accidentalmente l'operatore di uguaglianza == con quello di assegnamento = Paradossalmente la cosa peggiore è che questo

scambio generalmente non causa un errore di sintassi: anzi, le istruzioni che contengono questi errori sono compilate correttamente, ed è solo in fase di esecuzione che possono rivelare l'errore logico, perché i risultati sono errati.

Sono due gli aspetti del C++ a causare questo problema. Il primo è che qualsiasi espressione che assume un valore può essere utilizzata nella parte decisionale di qualsiasi struttura di controllo. Se il valore che essa assume è 0, l'espressione viene trattata come falsa, in tutti gli altri casi come vera. Il secondo è che gli assegnamenti in C++ assumono un valore, ovvero il valore assegnato alla variabile che si trova a sinistra dell'operatore. Per esempio supponiamo di voler scrivere

```
if ( payCode == 4 )
   cout << "You get a bonus!" << endl;
```

ma scriviamo accidentalmente

```
if ( payCode = 4 )
   cout << "You get a bonus!" << endl;
```

La prima istruzione `if` premia correttamente con un bonus una persona se ha `paycode` uguale a 4. La seconda `if`, quella con l'errore di battitura, effettua prima l'assegnamento del valore 4 a `paycode` e poi verifica se il valore dell'intera espressione è nullo, cioè se la condizione è vera o falsa. Nel nostro caso il valore è 4, per cui l'espressione è interpretata come vera. Ma qualsiasi valore non nullo viene interpretato come vero, per cui l'istruzione `if` adesso risulta sempre vera e la persona riceve sempre il bonus, indipendentemente da quanto vale `paycode`! Tra l'altro `paycode` è stato modificato, mentre si intendeva soltanto metterlo a confronto con un altro valore!

Errore comune 2.26

Se utilizzate l'operatore == in un assegnamento, o l'operatore = in un'uguaglianza, commettete un errore logico.

Collaudo e messa a punto 2.1

I programmatori scrivono normalmente condizioni del tipo `x == 7` *con il nome della variabile a sinistra e la costante a destra. Se rovesciate l'ordine, scrivendo* `7 == x`*, il compilatore vi avvertirà se scambiate accidentalmente l'operatore di uguaglianza con quello di assegnamento. Infatti scrivere* `7 = x` *è un errore di sintassi, perché a sinistra di un assegnamento va sempre la variabile. Questo piccolo stratagemma può aiutarvi a evitare errori logici potenzialmente disastrosi.*

I nomi delle variabili sono detti anche *lvalue*, dall'inglese *left value*, valore a sinistra dell'operatore di assegnamento. In modo simile le costanti sono dette *rvalue*, da right value, valore a destra dell'operatore di assegnamento. Gli lvalue possono essere utilizzati come rvalue, ma non vale il viceversa.

Torniamo al nostro argomento, la confusione tra i due operatori. Neanche la confusione opposta è così clemente. Supponiamo di voler assegnare un valore a una variabile con una semplice istruzione del tipo

```
x = 1;
```

ma di aver scritto invece

```
x == 1;
```

Neanche qui avremo un errore di sintassi. Ciò che accade è che il compilatore verifica l'espressione condizionale e basta. Se `x` vale `1`, la condizione è vera e l'espressione restituisce il valore `true`, altrimenti la condizione è falsa e l'espressione restituisce il valore `false`. Indipendentemente dal valore restituito però, non viene effettuato alcun assegnamento: il valore `1` viene semplicemente perduto, lasciando inalterata la variabile `x`. In fase di esecuzione questo comportamento, senza troppo significato, lascerà certamente il suo segno. Purtroppo, però, non ci viene in mente uno stratagemma da proporvi per scongiurare un errore di questo tipo!

Collaudo e messa a punto 2.2

Con il vostro editor di testo cercate tutte le occorrenze dell'operatore = all'interno dei vostri programmi e verificate che sia utilizzato sempre correttamente.

2.21 Riepilogo dei concetti fondamentali della programmazione strutturata

I programmatori assomigliano un po' agli architetti, i quali progettano le loro costruzioni attingendo al sapere collettivo della loro professione. Il nostro campo è molto più giovane dell'architettura, e il sapere collettivo decisamente meno diffuso. Fra le cose che abbiamo imparato c'è il fatto che la programmazione strutturata ci consente di scrivere programmi migliori: essi sono più semplici da capire, verificare e da modificare, e possiamo facilmente effettuarne il debugging e dimostrarne la correttezza funzionale, in senso matematico.

La Figura 2.32 riepiloga le strutture di controllo del C++. I cerchietti in figura indicano i singoli punti d'entrata e d'uscita di ciascuna struttura. Connettendo arbitrariamente i *simboli in un diagramma di flusso si finisce per scrivere programmi non strutturati. Di* conseguenza i professionisti della programmazione hanno scelto di combinare i vari simboli in modo da formare un insieme limitato di strutture di controllo: la scrittura di programmi strutturati si riduce quindi alla combinazione di queste strutture di controllo in due semplici modi.

Per semplicità si è scelto di utilizzare solo strutture di controllo a una sola entrata e una sola uscita. Il compito di connettere le varie strutture di controllo risulta così semplificato: l'uscita di una struttura viene collegata all'entrata della successiva. In questo modo si "mettono in sequenza", cioè una dopo l'altra, le diverse strutture di controllo di un programma. È anche possibile nidificare le strutture di controllo, ponendone una all'interno dell'altra.

In Figura 2.33 elenchiamo le regole per formare in modo corretto un programma strutturato. È implicito che il simbolo rettangolo possa indicare qualsiasi operazione, incluso l'input/output.

Se applichiamo le regole in Figura 2.33 [illegible] un diagramma di flusso strutturato a blocchi. Per esempio, applicando ripetutamente la regola 2 al diagramma di flusso elementare avremo un diagramma contenente molti rettangoli in sequenza (Figura 2.35). Notate che la regola 2 genera una sequenza di strutture di controllo: possiamo chiamare la regola 2 regola della sequenza.

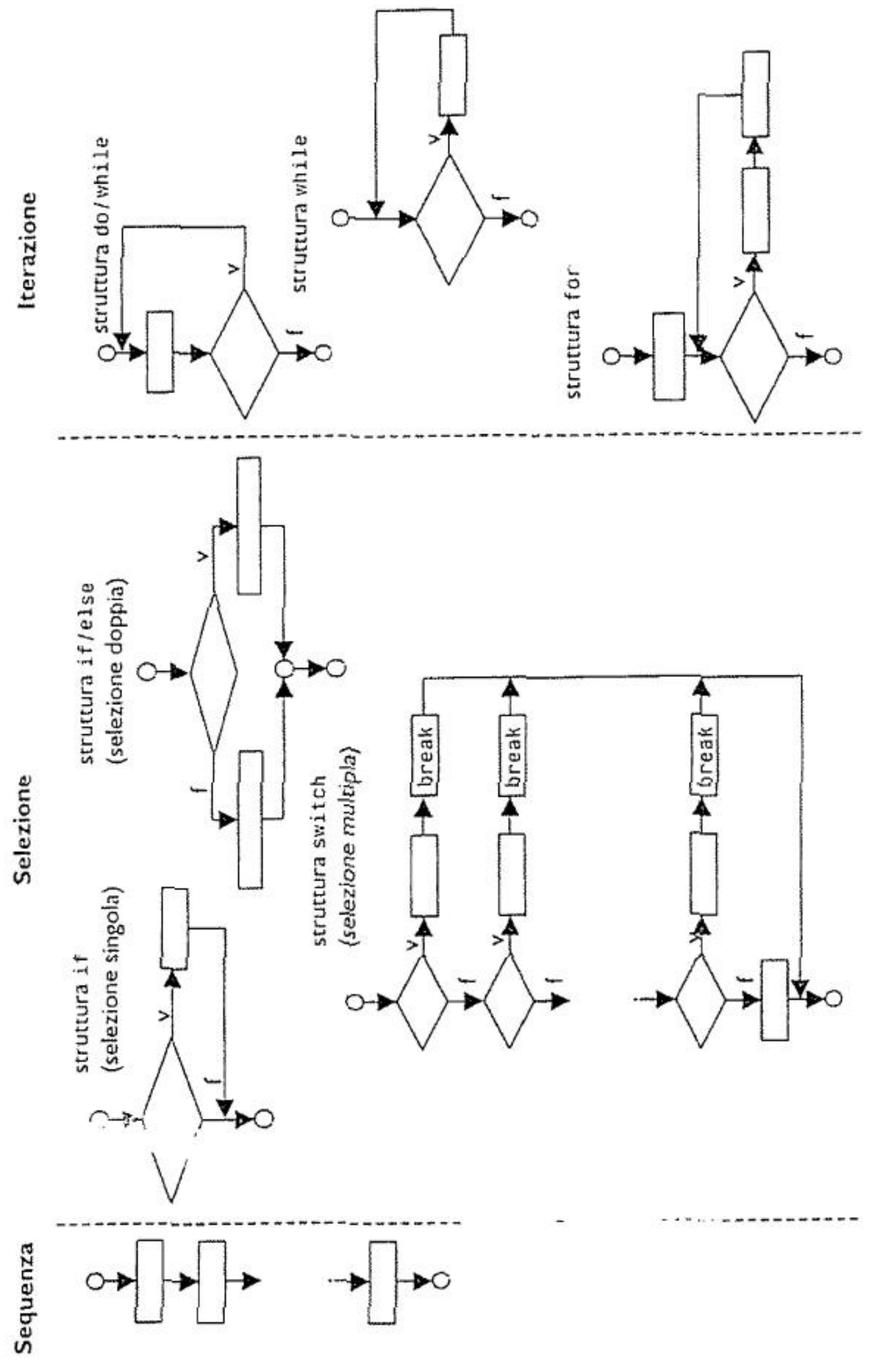


Figura 2.32 Le strutture del C++ hanno tutte una sola entrata e una sola uscita.

La regola 3 invece è la regola di nidificazione. Se applichiamo ripetutamente la regola 3 al diagramma di flusso elementare avremo un diagramma con diverse strutture di controllo nidificate. Per esempio, in Figura 2.36, il rettangolo del diagramma di flusso elementare viene sostituito una prima volta con una struttura di selezione doppia (`if/else`). Riapplichiamo poi la regola 3 a entrambi i rettangoli della struttura di selezione doppia, sostituendo ognuno di essi con strutture di selezione doppia. I rettangoli tratteggiati intorno a ogni struttura di selezione doppia ricordano il rettangolo originario del diagramma di flusso elementare.

La regola 4 genera strutture più complesse e con un livello di nidificazione più profondo. Il diagramma che abbiamo dopo aver applicato le regole in Figura 2.33 costituisce l'insieme di tutti i diagrammi di flusso possibili, e quindi di tutti i programmi strutturati possibili.

Regole per comporre programmi strutturati

1) Iniziare dal diagramma di flusso elementare (Figura 2.34).
2) Ogni rettangolo (azione) può essere sostituito da due rettangoli (azioni) in sequenza.
3) Ogni rettangolo (azione) può essere sostituito da qualsiasi struttura di controllo (sequenza, `if`, `if/else`, `switch`, `while`, `do/while` o `for`).
4) Le regole 2 e 3 possono essere applicate indefinitamente e in qualsiasi ordine.

Figura 2.33 Regole per comporre programmi strutturati.

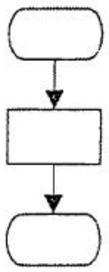


Figura 2.34 Il diagramma di flusso elementare.

La bellezza dell'approccio strutturato sta nella sua semplicità: tutto ciò che si può fare è combinare solo sette pezzi a una sola entrata e una sola uscita, assemblandoli in due modi possibili. La Figura 2.37 illustra il genere di blocchi che abbiamo se applichiamo la regola 2, cioè una sequenza di blocchi, e il genere di blocchi che abbiamo se applichiamo la regola 3, cioè dei blocchi nidificati. In figura mostriamo anche il genere di blocchi sovrapposti che è impossibile trovare nei programmi strutturati, a causa dell'eliminazione delle istruzioni `goto`.

Se si seguono le regole di Figura 2.33 non è possibile creare diagrammi di flusso non strutturati, come quello in Figura 2.38. Se non siete sicuri di aver creato un diagramma strutturato, applicate al contrario le regole di Figura 2.33, cercando di risalire dal vostro diagramma al diagramma elementare. Se ci riuscite, avete creato un diagramma di flusso strutturato, altrimenti no.

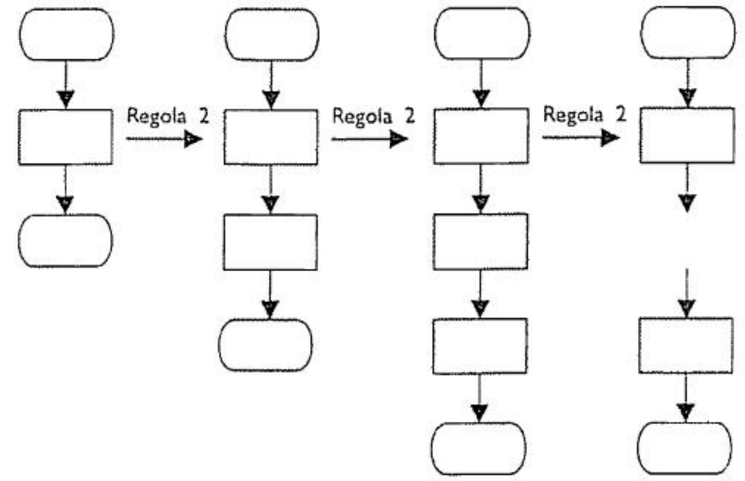


Figura 2.35 Applicazione ripetuta della regola 2 di Figura 2.33 al diagramma di flusso elementare.

La programmazione strutturata si accompagna alla semplicità. Böhm e Jacopini hanno dimostrato come siano possibili tre sole forme di controllo:

- Sequenza
- Selezione
- Iterazione

con le quali poter costruire *qualunque programma*.

Tralasciando la prima forma di controllo, poiché ovvia, la selezione si implementa in uno di questi tre modi:

- costrutto `if` (selezione singola)
- costrutto `if/else` (selezione doppia)
- costrutto `switch` (selezione multipla)

Possiamo dimostrare semplicemente che il solo `if` è sufficiente a formare qualsiasi forma di selezione. In sostanza possiamo fare tutto con un costrutto `if/else`, e possiamo implementare un costrutto `switch` combinando diverse `if`, anche se il codice non risulterà troppo leggibile.

L'iterazione si implementa in uno di questi tre modi:

- costrutto `while`
- costrutto `do/while`
- costrutto `for`

Possiamo dimostrare semplicemente che il solo `while` è sufficiente a formare qualsiasi forma di iterazione. In sostanza possiamo fare tutto con un costrutto `do/while`, e possiamo implementare un costrutto `for` tramite un costrutto `while`, anche se non con la stessa semplicità.

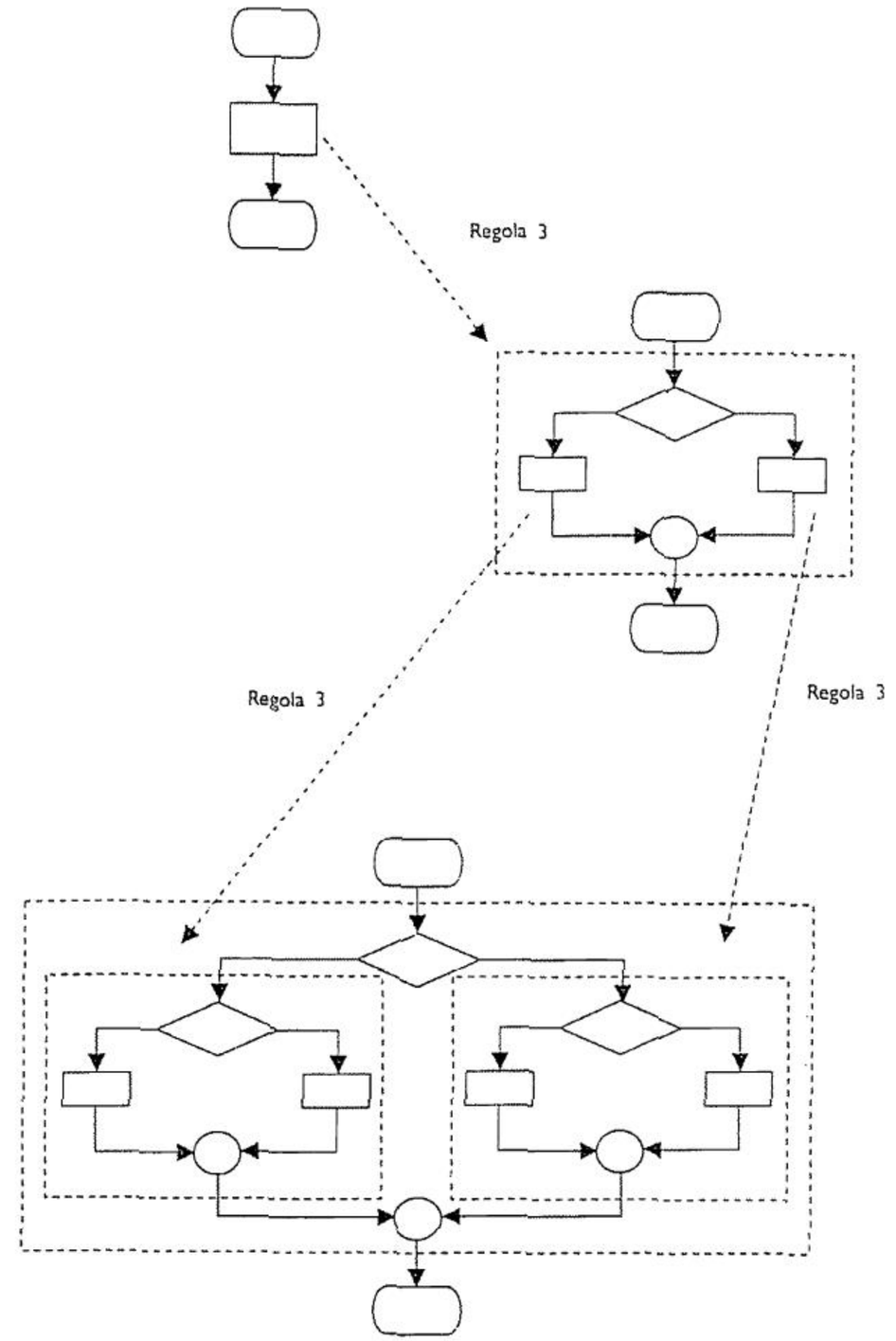


Figura 2.36 Applicazione della regola 3 di Figura 2.33 al diagramma di flusso elementare.

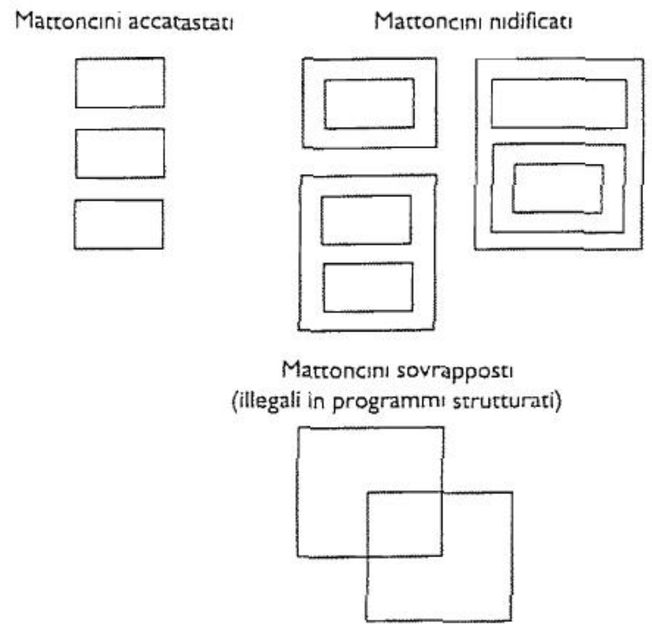


Figura 2.37 L'aspetto che assumono i blocchi se si mettono in sequenza, si nidificano o si sovrappongono.

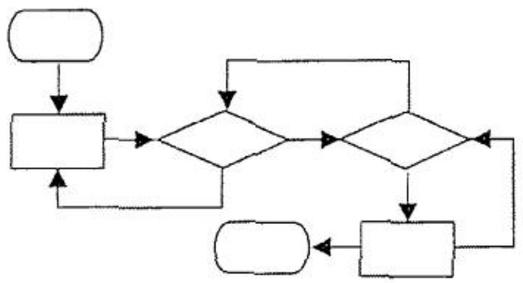


Figura 2.38 Diagramma di flusso non strutturato.

In questo capitolo abbiamo parlato della composizione di programmi a partire da strutture di controllo del tipo azione/decisione. Nel Capitolo 3 introdurremo la prossima unità di strutturazione di un programma, la funzione. Impareremo a scrivere programmi combinando diverse funzioni, e ognuna di esse sarà composta da strutture di controllo. Parleremo anche della capacità delle funzioni di rendere il software riutilizzabile. Nel Capitolo 6 introdurremo l'altra unità di strutturazione del C++, la classe. A partire dalle classi saremo in grado di creare degli oggetti, per cui cominceremo la nostra esplorazione della programmazione orientata agli oggetti. Per il momento continuiamo il nostro discorso introduttivo, introducendo un problema che dovrete risolvere con le tecniche OOD, ovvero di progettazione orientata agli oggetti.

2.22 Pensare in termini di oggetti: come individuare le classi in un problema [progetto opzionale]

Il caso di studio che proponiamo in questa sezione è opzionale ed è un esempio di progettazione e implementazione di un software orientato agli oggetti. Le sezioni "Pensare in termini di oggetti" che troverete d'ora in avanti a fine capitolo vi consentiranno di familiarizzare con il concetto di "progettazione orientata agli oggetti" (OOD) e verteranno sulla realizzazione di un simulatore software di un ascensore. Riteniamo che questo progetto sarà per voi un'esperienza importante, graduale e completa di progettazione e implementazione. Nei Capitoli dal 2 al 5 vi condurremo attraverso i vari passi della progettazione orientata agli oggetti utilizzando l'UML, mentre nei Capitoli 6, 7 e 9 implementeremo il codice del simulatore utilizzando le tecniche di programmazione orientata agli oggetti (OOP) del C++. Studieremo questo progetto in modo completo, fino alla sua soluzione, perciò esitiamo a definirlo soltanto un esercizio, perché è piuttosto un piccolo corso completo che ci porterà fino all'analisi dettagliata del codice C++ finale. Confidiamo che questa esperienza possa darvi un'idea della tipologia dei reali problemi dell'industria.

2.22.1 Definizione del problema

Un'azienda intende costruire un edificio di due piani da adibire a uso uffici dotato di un ascensore. L'azienda vi affida lo sviluppo di un *simulatore software* dell'ascensore orientato agli oggetti e in C++, che rappresenti un modello operativo dell'ascensore utile per determinare se questo si adatta effettivamente alle loro esigenze.

Il simulatore include un timer che viene azzerato all'inizio della giornata e incrementato ad istanti regolari. Il simulatore dovrebbe anche includere un meccanismo di gestione degli arrivi che all'inizio della giornata fornisca due istanti temporali in modo casuale: il primo rappresenta l'istante in cui una persona arriverà al primo piano e premerà il pulsante esterno che chiama l'ascensore, il secondo l'istante in cui ne arriverà una al secondo piano. Questi due istanti temporali sono interi casuali nell'intervallo 5-20 (estremi compresi). [Nota: imparerete a generare interi casuali nel Capitolo 3]. Quando il timer raggiunge il primo di questi istanti, il meccanismo di gestione degli arrivi crea una persona, essa raggiunge l'ascensore al piano appropriato e preme il pulsante esterno dell'ascensore. [Nota: se gli istanti assegnati alle due persone sono identici esse arriveranno all'ascensore nel loro piano contemporaneamente]. Il pulsante esterno si illumina, indicando che è stato premuto. [Nota: l'illuminazione del pulsante si verifica automaticamente quando questo viene premuto e non necessita di programmazione; la lampadina interna del pulsante si spegne automaticamente quando il pulsante viene rilasciato]. All'inizio della giornata l'ascensore sta a porte chiuse al primo piano, e per risparmiare energia, esso si sposta solo quando è necessario. I possibili movimenti sono solamente due, in su e in giù, alternativamente.

Per semplicità supporremo che l'ascensore e i piani abbiano una capacità massima di una sola persona. Il meccanismo di gestione degli arrivi deve verificare innanzitutto che il piano sia libero, prima di creare una persona su di esso. Se il piano è occupato, il meccanismo di gestione degli arrivi rimanda questa operazione all'istante successivo, dando l'opportunità all'ascensore di prelevare nel frattempo la persona e liberare il piano. Non appena una persona è posta su un piano, il meccanismo di simulazione genera il successivo istante casuale (da 5 a 20 secondi nel futuro) in cui far arrivare la prossima persona su quel piano.

Quando l'ascensore arriva al piano, rilascia il pulsante interno e attiva il campanello d'avviso che si trova al suo interno, quindi segnala la sua presenza su quel piano. Il piano, in risposta, rilascia il pulsante esterno dell'ascensore e accende la spia verde che ne indica la presenza. A quel punto l'ascensore apre le porte. [Nota: la porta del piano si apre automaticamente insieme con quella interna dell'ascensore e non necessita di programmazione]. Se nell'ascensore c'è un passeggero, egli scende.

Una persona entra nell'ascensore e preme il pulsante interno, che si illumina (automaticamente, senza necessità di programmazione) in quel momento e si spegne quando l'ascensore arriva al piano, rilasciando anche il pulsante interno. [Nota: dal momento che i piani sono solo due, serve un solo pulsante interno, che dice semplicemente all'ascensore di andare all'altro piano]. A quel punto l'ascensore chiude le porte e comincia la corsa verso l'altro piano. Quando arriva, se non entra nessuno e se (nel frattempo) non è stato premuto il pulsante esterno dell'altro piano, l'ascensore chiude le porte e rimane su quel piano in attesa finché non viene premuto il pulsante esterno su uno dei piani.

Per semplicità assumeremo che tutte le attività che si verificano dalla fine della corsa alla chiusura delle porte impieghino un tempo nullo. [Nota: questo non cambia il fatto che si verificano comunque in una data sequenza, ad esempio l'ascensore deve aprire le porte prima che il passeggero possa scendere]. Il tempo impiegato dall'ascensore per raggiungere l'altro piano è sempre pari a cinque secondi. Ad ogni istante, il simulatore fornisce l'orario attuale al meccanismo di gestione degli arrivi e all'ascensore. Essi utilizzano questo valore per determinare quali sono le azioni da intraprendere in quel momento particolare, per esempio il meccanismo di gestione degli arrivi per determinare se debba creare una persona, e l'ascensore, se è in moto, per verificare che sia arrivato al piano di destinazione.

Il simulatore visualizzerà dei messaggi sullo schermo che descrivono le attività che si verificano nel sistema: ad esempio una persona che preme il pulsante esterno in un piano, l'ascensore che arriva al piano di destinazione, il tic del timer, una persona che entra nell'ascensore, e così via. L'output avrà questo aspetto:

```
Enter run time: 30
(scheduler schedules next person for floor 1 at time 5)
(scheduler schedules next person for floor 2 at time 17)
*** ELEVATOR SIMULATION BEGINS ***
TIME: 1
elevator at rest on floor 1
TIME: 2
elevator at rest on floor 1
TIME: 3
elevator at rest on floor 1
TIME: 4
elevator at rest on floor 1
```

```
TIME: 5
scheduler creates person 1
person 1 steps onto floor 1
person 1 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 20)
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 1 enters elevator from floor 1
person 1 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
elevator begins moving up to floor 2 (arrives at time 10)
TIME: 6
elevator moving up
TIME: 7
elevator moving up
TIME: 8
elevator moving up
TIME: 9
elevator moving up
TIME: 10
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 1 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator at rest on floor 2
TIME: 11
elevator at rest on floor 2
TIME: 12
elevator at rest on floor 2
TIME: 13
elevator at rest on floor 2
TIME: 14
elevator at rest on floor 2
TIME: 15
elevator at rest on floor 2
TIME: 16
elevator at rest on floor 2
TIME: 17
```



```
scheduler creates person 2
person 2 steps onto floor 2
person 2 presses floor button on floor 2
floor 2 button summons elevator
(scheduler schedules next person for floor 2 at time 34)
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 2 enters elevator from floor 2
person 2 presses elevator button
elevator button tells elevator to prepare to leave
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1 (arrives at time 22)
TIME: 18
elevator moving down
TIME: 19
elevator moving down
TIME: 20
scheduler creates person 3
person 3 steps onto floor 1
person 3 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 26)
elevator moving down
TIME: 21
elevator moving down
TIME: 22
elevator arrives on floor 1
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 2 exits elevator on floor 1
person 3 enters elevator from floor 1
person 3 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
elevator begins moving up to floor 2 (arrives at time 27)
TIME: 23
elevator moving up
TIME: 24
elevator moving up
TIME: 25
elevator moving up
```

```
TIME: 26
scheduler creates person 4
person 4 steps onto floor 1
person 4 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 35)
elevator moving up
TIME: 27
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 3 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1 (arrives at time 32)
TIME: 28
elevator moving down
TIME: 29
elevator moving down
TIME: 30
elevator moving down
*** ELEVATOR SIMULATION ENDS ***
```

Il nostro obiettivo nelle sezioni "Pensare in termini di oggetti" è implementare un simulatore software che rappresenti operativamente l'ascensore per un numero di secondi deciso dall'utente.

2.22.2 Analisi e progettazione del sistema

In questa sezione e nelle prossime attraverseremo le fasi di progettazione orientata agli oggetti del sistema ascensore. L'UML è stato progettato per essere utilizzato in qualsiasi processo di analisi e progettazione orientati agli oggetti (OOAD, ne esistono diversi). In questo caso di studio utilizzeremo una nostra versione semplificata di un processo di progettazione orientato agli oggetti.

Prima di iniziare dobbiamo esaminare la natura delle simulazioni. Una simulazione si compone di due parti. La prima contiene tutti gli elementi che appartengono al sistema reale che vogliamo simulare, nel nostro caso per esempio l'ascensore, i piani, i pulsanti, le spie, ecc. Chiameremo questa prima parte la *parte del mondo reale*. L'altra parte contiene tutti gli elementi che sono indispensabili per simulare tale sistema reale, come il timer e il programmatore. Chiameremo questa parte la *parte di controllo*. Cercheremo di tenere a mente questa distinzione nella progettazione del nostro sistema.

2.22.3 I diagrammi dei casi d'uso

Quando un programmatore inizia a lavorare su un progetto raramente si ritrova subito con una definizione del problema dettagliata, come quella che vi abbiamo fornito all'inizio della sezione (Sezione 2.22). Questo documento e altri simili sono normalmente il risultato della fase di *analisi orientata agli oggetti* (OOA). In questa fase si parla con le persone che hanno commissionato il progetto e con quelle che alla fine lo utilizzeranno. Si utilizzano quindi le informazioni raccolte nei colloqui per compilare un elenco di *requisiti del sistema*, che costituiranno la traccia da seguire durante la progettazione del sistema. Nel nostro caso la definizione del problema contiene già i requisiti di sistema dell'ascensore. Il risultato della fase di analisi deve specificare in modo chiaro *che cosa* deve fare il sistema, mentre il risultato della fase di progettazione deve specificare *come* deve essere costruito per compiere le operazioni desiderate.

L'UML prevede il *diagramma dei casi d'uso*, che semplifica proprio la stesura dei requisiti. Il diagramma dei casi d'uso rappresenta le interazioni tra i clienti esterni al sistema e i *casi d'uso* del sistema. Ogni caso d'uso rappresenta una diversa funzionalità che il sistema dovrà fornire al cliente; per esempio, uno sportello bancario automatico ha diversi casi d'uso, come per esempio "Deposito" "Prelievo" e "Trasferimento di valuta"

La Figura 2.39 mostra il diagramma dei casi d'uso relativo al nostro ascensore. La figura a bastoncini è detta *attore*. Gli attori sono entità esterne che utilizzano il nostro sistema, come le persone, i robot o altri sistemi. Gli unici attori del nostro sistema sono le persone che richiedono una corsa dell'ascensore, perciò rappresenteremo un solo attore e lo chiameremo "Persona" Il nome dell'attore compare sotto la figura a bastoncini.

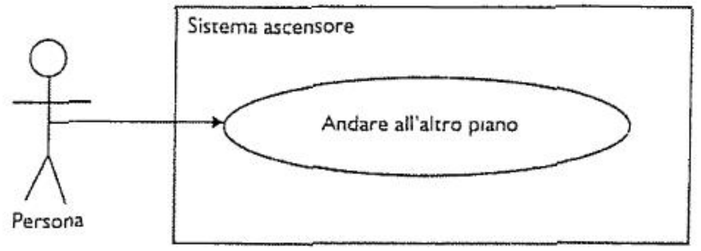


Figura 2.39 Diagramma dei casi d'uso relativo al sistema ascensore.

Il *system box* (ovvero il rettangolo che racchiude il sistema in figura) contiene i casi d'uso del sistema. Notate l'etichetta del rettangolo "Sistema ascensore" questo titolo indica che *questo modello dei casi d'uso si concentra sui comportamenti del sistema che vogliamo simulare* (ovvero "ascensore che trasporta persone"), *rispetto ai comportamenti della simulazione vera e propria* (ovvero "creare le persone e programmare gli arrivi").

L'UML rappresenta ogni caso d'uso con un ovale. Nel nostro semplice sistema gli attori utilizzano l'ascensore al solo scopo di andare all'altro piano, cioè il sistema fornisce una sola funzionalità agli utenti. L'unico caso d'uso presente sarà quindi "Andare all'altro piano"

Man mano che si procede nella progettazione del sistema, è utile far riferimento al diagramma dei casi d'uso per essere sicuri di considerare tutte le richieste dei clienti. Il nostro progetto ne contiene soltanto uno, ma nei sistemi più complessi tali diagrammi

sono strumenti indispensabili, in quanto aiutano i progettisti a non perdere di vista nessuna funzionalità. Lo scopo di un diagramma dei casi d'uso è di evidenziare le tipologie di interazioni che gli utenti hanno con il sistema, senza scendere troppo nel dettaglio.

2.22.4 L'identificazione delle classi di un sistema

Il passo successivo del nostro procedimento OOD consiste nell'identificare le classi del problema. Arriveremo alla fine a descrivere queste classi in modo formale e a implementarle in C++. Per prima cosa torniamo alla definizione del problema e cerchiamo tutti i nomi presenti nel testo: probabilmente questi rappresentano la maggior parte delle classi (o di loro istanze) necessarie per implementare il simulatore; essi sono riportati in Figura 2.40.

Elenco dei nomi presenti nella definizione del problema

azienda (company)
edificio (building)
ascensore (elevator)
simulatore (simulator)
timer (clock)
istante temporale (time)
programmatore (scheduler)
persona (person)
piano I (floor 1)
pulsante esterno (floor button)
piano II (floor 2)
porta dell'ascensore (elevator door)
energia (energy)
capacità (capacity)
pulsante interno (elevator button)
campanello (elevator bell)
spia di presenza dell'ascensore
persona in attesa al piano
passeggero

Figura 2.40 Elenco dei nomi presenti nella definizione del problema.

Selezioniamo ora solo i nomi che compiono funzioni importanti nel nostro sistema. Per questo motivo omettiamo i seguenti:

- azienda
- simulatore
- istante temporale
- energia
- capacità

Non abbiamo bisogno di rappresentare l'"azienda" con una classe, perché non fa parte *della simulazione, vi ha semplicemente commissionato il simulatore. Il "simulatore"* è l'intero programma in C++, non una singola classe. La voce "istante temporale" è una proprietà del timer, non un'entità a sé stante. Non rappresentiamo neanche la voce "energia", benché le compagnie dell'elettricità, del gas o di combustibili potrebbero volerle nei loro programmi di simulazione, ed infine notiamo che anche "capacità" è una proprietà dell'ascensore e del piano, non un'entità a sé stante.

Determiniamo ora le classi del nostro sistema raggruppando i nomi che sono rimasti in categorie. Ciascun nome ancora presente si riferisce a una o più delle seguenti categorie:

- edificio
- ascensore
- timer
- meccanismo di gestione degli arrivi
- persona (persona in attesa al piano, passeggero)
- piano (piano I, piano II)
- *pulsante esterno*
- pulsante interno
- campanello
- spia
- porta

Probabilmente queste categorie sono le classi che dovremo implementare per il nostro sistema. Come vedete, abbiamo creato una categoria per i pulsanti esterni dell'ascensore e un'altra distinta per quello interno. Questi due tipi di pulsante effettuano compiti diversi nella simulazione, perché quelli esterni chiamano l'ascensore, mentre quello interno gli ordina di effettuare una corsa verso l'altro piano.

Adesso possiamo rappresentare le classi del sistema sulla base delle categorie che abbiamo appena individuato. Per convenzione la prima lettera del nome di tutti le classi sarà maiuscola e, se il nome di una classe è composto da più parole, eliminiamo gli spazi e rendiamo maiuscola la prima lettera di ogni parola (per es. `NomeDiPiuParole`). Seguendo questa convenzione, definiamo le classi `Elevator` (ascensore), `Clock` (timer), `Scheduler` (meccanismo di gestione degli arrivi), `Person` (persona), `Floor` (piano), `Door` (porta), `Building` (edificio), `FloorButton` (pulsante esterno), `ElevatorButton` (pulsante interno), `Bell` (campanello) e `Light` (spia). Costruiremo il nostro sistema utilizzando queste classi come mattoni fondamentali, ma prima di addentrarci è bene comprendere meglio le loro relazioni reciproche.

2.22.5 I diagrammi delle classi

UML consente di rappresentare le classi di un sistema e le rispettive relazioni tramite il *diagramma delle classi*, di cui vedete un esempio in Figura 2.41. La prima classe che andiamo a definire è `Elevator`. In questo tipo di diagramma, le classi sono rappresentate con un rettangolo suddiviso in tre parti: la parte superiore contiene il nome della classe, quella centrale gli *attributi* della classe (che studieremo nel Capitolo 3), mentre quella inferiore le *operazioni* della classe (che studieremo nel Capitolo 4).

Le classi sono in relazione l'una con l'altra per mezzo di *associazioni*. La Figura 2.42 mostra le relazioni delle classi `Building`, `Elevator` e `Floor`. Come notate, i rettangoli in questo diagramma non sono stati suddivisi: UML consente di abbreviare i simboli delle classi in questo modo, consentendo di creare diagrammi più leggibili.

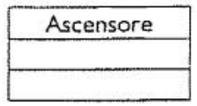


Figura 2.41 Rappresentazione di una classe in UML.

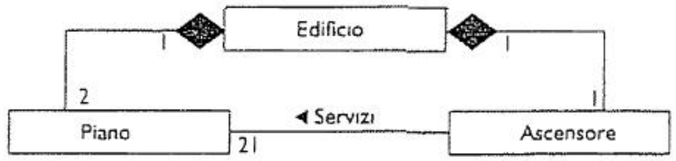


Figura 2.42 Associazioni in un diagramma delle classi.

In questo diagramma delle classi, un'associazione è rappresentata con una linea continua che connette due classi. I numeri adiacenti alle linee esprimono i valori di *molteplicità*, ovvero indicano *quanti* oggetti di una classe partecipano all'associazione. Dal diagramma vediamo che due oggetti della classe `Floor` partecipano all'associazione con un oggetto della classe `Building`, perciò la classe `Building` ha una relazione *uno-a-due* con la classe `Floor`: possiamo anche invertire l'espressione, dicendo che la classe `Floor` ha una relazione *due-a-uno* con la classe `Building`. Dal diagramma vediamo poi che la classe `Building` ha una relazione *uno-a-uno* con la classe `Elevator` e viceversa. Grazie a UML possiamo indicare diversi tipi di molteplicita. La Figura 2.43 illustra i tipi di molteplicità e mostra come si rappresentano.

È possibile dare un nome alle associazioni: ad esempio, la parola "Servizi" sulla linea che connette le classi `Floor` e `Elevator` e il nome di tale associazione (la freccia ne indica la direzione). Questa parte del diagramma si legge: "un oggetto della classe `Elevator` serve due oggetti della classe `Floor`"

Il rombo pieno unito alle linee di associazione della classe `Building` indica che la classe `Building` è in relazione di *composizione* con le classi `Floor` e `Elevator`: la composizione è una relazione del tipo *parte-di*. La classe che ha il simbolo di composizione (il rombo pieno) alla fine della linea di associazione è il "tutto" (nel nostro caso `Building`), mentre la classe all'altra estremità della linea di associazione è la "parte" (`Floor` ed `Elevator`). [Nota: secondo le specifiche dell'UML 1.3, le classi di una composizione devono osservare le seguenti tre proprietà: 1) solo una classe può rappresentare il "tutto" (ovvero il rombo può trovarsi su una sola estremità della linea di associazione; 2) la composizione implica che le linee di vita delle parti coincidano con quella del tutto, e il tutto è responsabile della creazione e della distruzione delle proprie parti; 3) una parte può appartenere a un solo "tutto" per volta, anche se può essere rimossa e associata a un altro "tutto", che si assume la responsabilità della parte.]

Simbolo	Significato
0	Nessuna.
1	Uno.
m	Un valore intero.
0..1	Zero o uno.
m..*n*	Almeno m, ma non superiore a n.
*	Qualsiasi interno non negativo.
0..*	Zero o superiore
1..*	Uno o superiore

Figura 2.43 Tabella di molteplicità

La Figura 2.44 mostra il diagramma delle classi completo del nostro progetto. Vi sono tutte le classi che abbiamo individuato e le loro associazioni. [Nota: nel Capitolo 9 espanderemo questo diagramma utilizzando il concetto di *ereditarietà* della programmazione orientata agli oggetti].

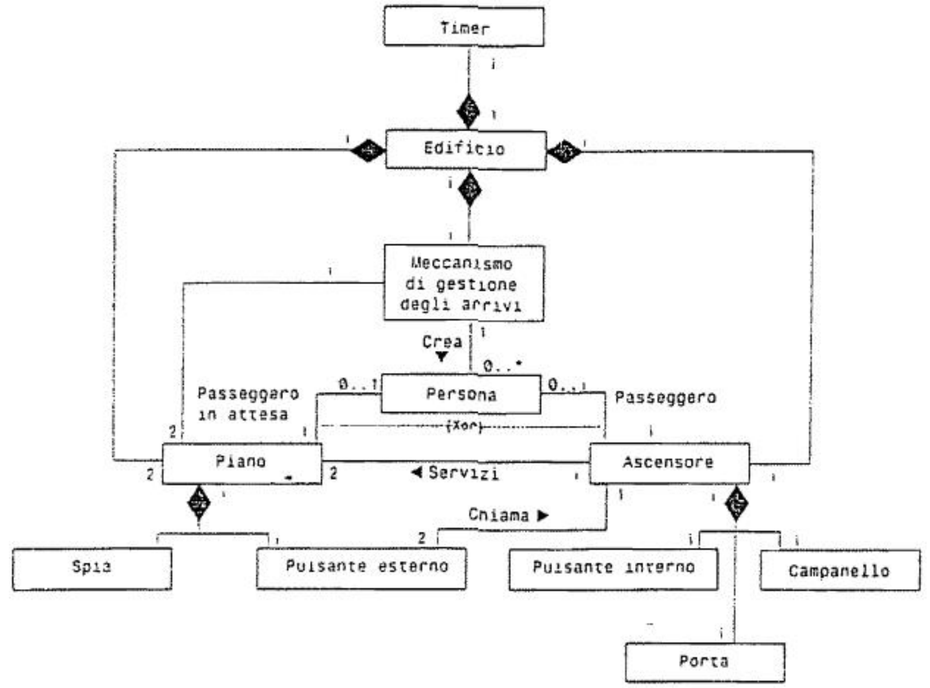


Figura 2.44 Diagramma delle classi completo del simulatore di ascensore

La classe `Building` si trova in cima al diagramma ed è composta da quattro classi (`Clock`, `Scheduler`, `Elevator` e `Floor`). Le classi `Clock` e `Scheduler` formano la parte di controllo della simulazione [Nota: la relazione di composizione tra `Building` e le classi `Clock` e `Scheduler` rappresenta una decisione di progettazione del tutto nostra. Conside-

riamo la classe `Building` sia come parte "del mondo reale" che "di controllo" della nostra simulaziona. Nella nostra progettazione affidiamo all'edificio la responsabilità di eseguire la simulazione.1. Notate la relazione uno-a-due tra `Building` e `Floor`.

La classe `Floor` è composta da un oggetto della classe `Light` e uno della classe `FloorButton`. La classe `Elevator` è composta da un oggetto della classe `ElevatorButton`, uno della classe `Door` e uno della classe `Bell`.

Le classi coinvolte in un'associazione possono anche avere dei *ruoli*. I ruoli aiutano a chiarire la relazione che sussiste tra due classi. Per esempio `Person` interpreta il ruolo "passeggero in attesa" nella sua associazione con la classe `Floor` (perché la persona attende l'ascensore), e il ruolo "passeggero" nella sua associazione con la classe `Elevator`. In un diagramma delle classi il nome del ruolo svolto da una classe si scrive su uno dei due lati della linea di associazione, vicino al rettangolo della classe. In un associazione ogni classe può svolgere un ruolo diverso.

L'associazione tra `Floor` e `Person` indica che un oggetto della classe `Floor` può essere *in relazione con zero o un oggetto della classe* `Person`. *La classe* `Elevator` *è in relazione* a sua volta con zero o un oggetto della classe `Person`. La linea tratteggiata che raccorda queste due linee di associazione indica un *vincolo* sulla relazione tra le classi `Person`, `Floor` e `Elevator`: esso specifica che un oggetto della classe `Person` può partecipare a una relazione con un oggetto della classe `Floor` o con un oggetto della classe `Elevator`, ma non con entrambi contemporaneamente. La notazione che esprime questo concetto è la parola *xor* (or esclusivo) tra parentesi graffe [Nota: in un diagramma UML i vincoli si possono scrivere con quello che è noto come *Object Constraint Language* (OCL). L'OCL è stato creato per definire i vincoli di un sistema in un modo chiaramente definito. Per *saperne di più* andate all'indirizzo *www-4.ibm.com/software/ad/standards/ocl.html*.]. L'associazione tra `Scheduler` e `Person` indica che un oggetto della classe `Scheduler` crea zero o più oggetti della classe `Person`.

I diagrammi degli oggetti

UML definisce anche i *diagrammi degli oggetti*, che sono simili ai diagrammi delle classi tranne per il fatto che rappresentano oggetti e *collegamenti* (relazioni tra oggetti). Come nel caso delle classi, i diagrammi degli oggetti rappresentano la struttura di un sistema e mostrano un'istantanea della sua struttura durante il funzionamento: ciò fornisce informazioni su quali oggetti partecipano al sistema in un dato momento.

La Figura 2.45 rappresenta un'istantanea del sistema quando nell'edificio non c'è nessuno (ovvero in quel momento nel sistema non esistono oggetti della classe `Person`). I nomi degli oggetti si scrivono generalmente nella forma: `nomeOggetto : NomeClasse`. La prima parola del nome dell'oggetto non inizia per maiuscola, al contrario delle successive. Tutti i nomi degli oggetti in un diagramma degli oggetti sono sottolineati. Per alcuni oggetti del diagramma (per esempio per i `FloorButton`) abbiamo omesso il nome. In sistemi più complessi in cui il modello conterrebbe molti nomi, c'è il rischio di creare diagrammi congestionati e difficili da leggere. Se non sappiamo che nome assegnare a un particolare oggetto, o se non ne è necessario uno perché l'importante è il tipo dell'oggetto, possiamo tranquillamente ometterlo. In questo caso scriviamo semplicemente il segno di due punti e il nome della classe.

A questo punto abbiamo identificato le classi del nostro sistema, anche se potremmo individuarne altre nelle ultime fasi della progettazione, e ne abbiamo esaminato i casi d'uso. Nella sezione "Pensare in termini di oggetti" del Capitolo 3 faremo uso delle informazioni che abbiamo raccolto finora per analizzare come cambia il sistema nel tempo. In seguito saremo in grado di ricavare nuove informazioni e descrivere le classi molto più dettagliatamente.

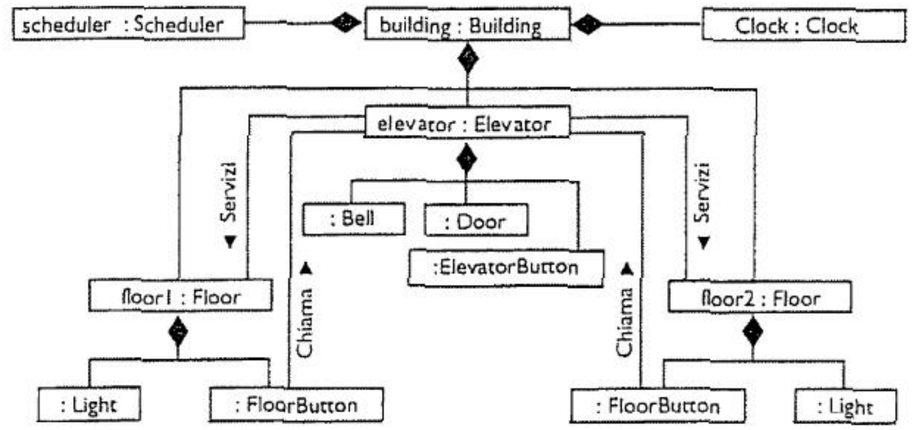


Figura 2.45 Diagramma degli oggetti di un edificio vuoto.

Esercizi di autovalutazione

Gli Esercizi dal 2.1 al 2.10 si riferiscono alle Sezioni 2.1- 2.12
Gli Esercizi dal 2.11 al 2.13 si riferiscono alle Sezioni 2.13 - 2.21.

2.1 Completate le seguenti affermazioni.
   a) *Tutti i programmi si possono scrivere* in termini di tre tipi di strutture di controllo: _____, _____ e _____
   b) La struttura di selezione _____ serve a eseguire un'operazione se una condizione è vera, e a eseguirne un'altra se è falsa.
   c) La ripetizione di un insieme di istruzioni un numero di volte determinato si chiama iterazione _____
   d) Se non si sa in anticipo quante volte verrà ripetuta un'istruzione, si può utilizzare un valore _____ per terminare l'iterazione.

2.2 Scrivete quattro istruzioni diverse che aggiungano 1 alla variabile intera `x`.

2.3 Scrivete le istruzioni in C++ che fanno ciò che segue:
   a) Assegnare la somma di `x` e `y` a `z` e incrementare il valore di `x` di 1 dopo aver effettuato il calcolo.
   b) Determinare se il valore della variabile `count` è maggiore di 10. Se la condizione risulta vera, visualizzare `"Il valore è maggiore di 10"`
   c) Decrementare la variabile `x` di 1 e sottrarla dalla variabile `totale`.
   d) Calcolare il resto della divisione di `q` per `divisor` e assegnare il risultato a `q`. Scrivete questa istruzione in due modi diversi.

2.4 Scrivete un'istruzione in C++ per ognuna delle seguenti operazioni:
a) Dichiarare le variabili `sum` e `x` di tipo `int`.
b) Inizializzare la variabile `x` a 1.
c) Inizializzare la variabile `sum` a 0.
d) Sommare la variabile `x` a `sum` e assegnare il risultato alla variabile `sum`.
Visualizzare `"La somma è: "`, seguito dal valore di `sum`.

2.5 Combinate le istruzioni che avete scritto nell'Esercizio 2.4 in un programma che calcola la somma dei numeri interi da 1 a 10. Utilizzate un ciclo `while` per il calcolo e le istruzioni di incremento. Il ciclo dovrebbe terminare quando il valore di `x` diventa 11.

2.6 Determinate il valore di ogni variabile dopo i seguenti calcoli. Per ipotesi, prima dell'esecuzione delle istruzioni, tutte le variabili valgono 5.
a) `product *= x++;`
b) `quotient /= ++x;`

2.7 Scrivete delle singole istruzioni che
a) Ricevono in input il valore della variabile `x` con `cin` e `>>`
b) Ricevono in input il valore della variabile `y` con `cin` e `>>`.
c) Inizializzano la variabile intera `i` a 1.
d) Inizializzano la variabile intera `power` a 1.
e) Moltiplicano la variabile `power` per `x` e assegnano il risultato a `power`.
f) Incrementano la variabile `y` di 1.
g) Determinano se `y` è minore o uguale a `x`.
h) Visualizzano la variabile `power` con `cout` e `<<`

2.8 Scrivete un programma che utilizza le istruzioni dell'Esercizio 2.7 per calcolare `x` elevato alla potenza `y`-esima. Il programma dovrebbe contenere la struttura di controllo iterativa `while`.

2.9 Trovate gli errori in questi segmenti di programma e cercate di correggerli:

a)
```
while ( c <= 5 )
      product *= c;
      ++c;
```
b)
```
cin << value;
```
c)
```
if ( gender == 1 )
   cout << "Woman" << endl;
else;
   cout << "Man" << endl;
```

2.10 Qualcosa non ci convince in questo costrutto `while`. Che cosa?

```
while ( z >= 0 )
  sum += z;
```

2.11 Stabilite quali affermazioni sono vere e quali false. Per le affermazioni false, datene una breve spiegazione.
a) La clausola `default` è obbligatoria in un costrutto `switch`.
b) L'istruzione `break` è obbligatoria nella clausola `default` di un costrutto `switch`, perché l'esecuzione della struttura di selezione termini correttamente.
c) L'espressione `( x > y && a < b )` è vera se è vera l'espressione `x > y` o se è vera l'espressione `a < b`.
d) Un'espressione che contiene l'operatore `||` è vera se è vero uno dei suoi operandi, o se sono veri entrambi.

2.12 Scrivete un'istruzione o un insieme di istruzioni per ognuna delle seguenti operazioni:
a) Sommare gli interi dispari tra 1 e 99 utilizzando un costrutto `for`. Per ipotesi, supponiamo di aver dichiarato le variabili `sum` e `count`.
b) Visualizzare il valore `333.546372` in un campo di `15` caratteri con una precisione di 1, 2 e 3 cifre. Visualizzate ogni numero sulla stessa riga e giustificatelo a sinistra all'interno del campo disponibile. Quali sono i tre valori visualizzati?
c) Calcolare il valore di `2.5` elevato alla terza utilizzando la funzione `pow`. Visualizzate il risultato con una precisione di 2 cifre in un campo di 10 posizioni. Qual è l'output di questa istruzione?
d) Visualizzare gli interi da 1 a 20 utilizzando un ciclo `while` con la variabile contatore `x`. Supponete di aver già dichiarato la variabile `x`, ma senza averla inizializzata. Visualizzate solo 5 interi per linea. Suggerimento: Utilizzate l'operazione `x % 5`. Se il suo valore è `0`, visualizzate un carattere di nuova linea, altrimenti un carattere di tabulazione.
e) Ripetete l'Esercizio 2.12 (d) utilizzando un costrutto `for`.

2.13 Trovate l'errore presente in ogni segmento di programma che vi proponiamo e spiegate come correggerlo.

a)
```
x = 1;
while ( x <= 10 );
   x++;
```
b)
```
for ( y = .1; y != 1.0; y += .1 )
   cout << y << endl;
```
c)
```
switch ( n ) {
case 1:
   cout << "The number is 1" << endl;
case 2:
   cout << "The number is 2" << endl;
   break;
default:
   cout << "The number is not 1 or 2" << endl;
   break;
}
```
d) Il codice seguente è stato scritto per visualizzare i numeri interi da 1 a 10.
```
n = 1;
while ( n < 10 )
   cout << n++ << endl;
```

Risposte agli esercizi di autovalutazione

2.1 a) Sequenziale, di selezione e di iterazione. b) `if/else`. c) controllato da contatore, o definito. d) Sentinella, segnale o flag.

2.2
```
1) x = x + 1;
2) x += 1;
3) ++x;
4) x++;
```

2.3 a)
```
z = x++ + y;
```
b)
```
if ( count > 10 )
   cout << "Count is greater than 10" << endl;
```
c)
```
total -= -x;
```
d)
```
q %= divisor;
q = q % divisor;
```

2.4
```
a) int sum, x;
b) x = 1;
c) sum = 0;
d) sum += x; or sum = sum + x;
   cout << "The sum is: " << sum << endl;
```

2.5 Come segue.

```
// Calcola la somma degli interi da 1 a 10
#include <iostream.h>

int main()
{
   int sum, x;
   x = 1;
   sum = 0;
   while ( x <= 10 ) {
      sum += x;
      ++x;
   }
   cout << "The sum is: " << sum << endl;
   return 0;
}
```

2.6
```
a) product = 25, x = 6;
b) quotient = 0, x = 6;
```

2.7
```
a) cin >> x;
b) cin >> y;
c) i = 1;
d) power = 1;
e) power *= x; o power = power * x;
f) y++;
g) if ( y <= x )
h) cout << power << endl;
```

2.8 Come segue.

```
// eleva  x alla y
#include <iostream.h>
int main()
{
   int x, y, i, power;

   i = 1;
   power = 1;
   cin >> x;
   cin >> y;

   while ( i <= y ) {
      power *= x;
      ++i;
   }

   cout << power << endl;
   return 0;
}
```


2.9
a) Errore: Manca la parentesi graffa chiusa nel corpo della `while`.
   Correzione: Scriverla dopo l'istruzione `++c;`.
b) Errore: Si utilizza l'inserzione nello stream anziché l'estrazione.
   Correzione: Al posto di `<<` scrivere `>>`
c) Errore: Il punto e virgola dopo la `else` causa un errore logico. Infatti la seconda istruzione di output sarà eseguita in ogni caso.
   Correzione: Eliminare il punto e virgola.

2.10 Il valore della variabile `z` non cambia mai nel corso della `while`. Quindi se la condizione di esecuzione del ciclo `( z >= 0 )` è vera, lo sarà sempre, causando un ciclo infinito. Occorre modificare il ciclo in modo che `z` possa alla fine diventare negativo.

2.11
a) Falsa. La clausola `default` è opzionale. Non occorre che essa sia definita se non c'è bisogno di un'azione di default.
b) Falsa. L'istruzione `break` viene utilizzata per uscire da un costrutto `switch`. Se la clausola `default` viene scritta per ultima, non occorre un'istruzione `break`.
c) Falsa. Entrambe le espressioni relazionali devono essere vere perché lo sia l'intera espressione, se si usa l'operatore `&&`.
d) Vera.

2.12 a)
```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
   sum += count;
```

b)
```
cout << setiosflags(ios::fixed | ios::showpoint | ios::left)
     << setprecision( 1 ) << setw( 15 ) << 333.546372
     << setprecision( 2 ) << setw( 15 ) << 333.546372
     << setprecision( 3 ) << setw( 15 ) << 333.546372
     << endl;
```

L'output è:

```
333.5          333.55         333.546
```

c)
```
cout << setiosflags( ios::fixed | ios::showpoint |
   << setprecision( 2 ) << setw( 10 ) << pow( 2.5, 3 )
   << endl;
```

L'output è:

```
     15.63
```

d)
```
x = 1;
while ( x <= 20 ) {
   cout << x;
   if ( x % 5 == 0 )
      cout << endl;
   else
      cout << '\t';
   x++;
}
```

e)
```
for ( x = 1; x <= 20; x++ ) {
   cout << x;
   if ( x % 5 == 0 )
      cout << endl;
   else
   cout << '\t';
}
```

oppure

```
for ( x = 1; x <= 20; x++ )
if ( x % 5 == 0 )
   cout << x << endl;
else
   cout << x << '\t';
```

2.13 a) Errore: Il punto e virgola dopo l'intestazione della `while` causa un ciclo infinito.
Correzione: Sostituire il punto e virgola con un { o eliminare sia il ; che la }.

b) Errore: Si utilizza un numero a virgola mobile per controllare il ciclo `for`.
Correzione: Utilizzare al suo posto un numero intero.

```
for ( y = 1; y != 10; y++ )
  cout << ( static_cast< float >( y ) / 10 ) << endl;
```

c) Errore: Manca l'istruzione `break` per la prima clausola `case`.
Correzione: Aggiungerlo. Notate che questo non è necessariamente un errore, perché il programmatore *può volere che ogni volta che si verifica il* `case 1` venga eseguito anche il `case 2`.

d) Errore: Si utilizza un operatore relazionale inappropriato nella condizione del costrutto `while`.
Correzione: Utilizzare `<=` al posto di `<` o cambiare `10` in `11`.

Esercizi

Gli Esercizi dal 2.14 al 2.38 si riferiscono alle Sezioni dalla 2.1 alla 2.12. Gli Esercizi dal 2.39 al 2.63 si riferiscono alle Sezioni dalla 2.13 alla 2.21.

2.14 Identificate e correggete gli errori presenti nei seguenti segmenti di codice (Nota: ne sono possibili anche più di uno per segmento):

a)
```
if ( age >= 65 );
   cout << "Age is greater than or equal to 65" << endl;
else
   cout << "Age is less than 65 << endl";
```

b)
```
if ( age >= 65 )
   cout << "Age is greater than or equal to 65" << endl;
else;
   cout << "Age is less than 65 << endl",
```

c)
```
int x = 1, total;
while ( x <= 10 ) ;
   total += x;
   ++x;
}
```

d)
```
While ( x <= 100 )
   total += x;
   ++x;
```

e)
```
while ( y > 0 ) ;
   cout << y << endl;
   ++y;
}
```

2.15 Che cosa visualizza il seguente programma?

```
#include <iostream.h>
int main()
{
   int y, x = 1, total = 0;
   while ( x <= 10 ) {
      y = x * x;
      cout << y << endl;
      total += y;
      ++x;
   }
   cout << "Total is " << total << endl;
   return 0;
}
```


Per gli Esercizi dal 2.16 al 2.19 seguite questa procedura:

a) Leggete la definizione del problema.
b) Formulate l'algoritmo con lo pseudocodice e la metodologia di raffinamento top-down.
c) Scrivete il vostro programma in C++.
Eseguite e verificate la correttezza del vostro programma.

2.16 Gli automobilisti controllano spesso il contachilometri. Un automobilista ha mantenuto una sorta di diario di bordo segnando tutti i pieni di benzina effettuati, i chilometri percorsi e i litri di benzina per ogni pieno. Scrivete un programma in C++ che riceve in input i chilometri e i litri di benzina per ogni pieno. Lo scopo del programma è di calcolare il rapporto chilometri/litro di benzina *relativo a ogni pieno*.

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1

The overall average miles/gallon was 21.601423
```

2.17 Scrivete un programma in C++ che determina se un cliente di un grande magazzino ha esaurito il suo credito. Per ogni cliente il programma ha a disposizione questi dati:

a) Numero del conto cliente (un intero)
b) Bilancio all'inizio del mese
c) Totale degli acquisti del cliente nel mese corrente
d) Totale dei crediti applicati al conto cliente nel mese corrente
e) Credito massimo consentito

Il programma dovrebbe richiedere in input queste grandezze, calcolare il nuovo bilancio del conto (= bilancio iniziale + acquisti − crediti) e determinare se il nuovo bilancio supera il credito massimo concesso al cliente. Se un cliente ha superato il suo credito massimo, il programma dovrebbe segnalarlo con un messaggio.

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
```

```
Account:      100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter account number (-1 to end): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00

Enter account number (-1 to end): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00

Enter account number (-1 to end): -1
```

2.18 Una grande multinazionale del settore chimico retribuisce i suoi rappresentanti con commissioni. Ogni rappresentante riceve un fisso di $200 a settimana più il 9% di commissione sulle vendite all'ingrosso di ogni settimana. Per esempio, un rappresentante che vende per un valore di $5000 di prodotti chimici, riceve $200 più il 9% di 5000, per un totale di $650. Scrivete un programma che richieda in input l'ammontare delle vendite all'ingrosso di un rappresentante e calcoli il guadagno del rappresentate per la settimana appena trascorsa. Elaborate i dati di un rappresentante alla volta.

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 6000.00
Salary is: $740.00

Enter sales in dollars (-1 to end): 7000.00
Salary is: $830.00

Enter sales in dollars (-1 to end): -1
```

2.19 Scrivete un programma che determini il salario lordo di un impiegato. La compagnia retribuisce gli impiegati con un fisso orario per le prime 40 ore, e con un fisso orario maggiorato del 50% per ogni ora di straordinario a partire dalla 41-esima. Supponete di ricevere una lista di impiegati, comprensiva del numero di ore di presenza e della paga oraria base di ciascun impiegato. Il vostro programma dovrebbe richiedere in input le informazioni relative a ciascun impiegato e visualizzarne il salario lordo.

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter hours worked (-1 to end): -1
```


2.20 Trovare il più grande di una serie di numeri è un problema frequente in molte applicazioni informatiche. Per esempio, un programma che determina il vincitore di un concorso aziendale di vendita deve determinare il maggior numero di unità vendute da ciascun rappresentante. Il rappresentante che ha venduto più unità vince. Scrivete lo pseudocodice e il codice C++ di un programma che richiede in input una serie di 10 valori e ne determina il maggiore. Suggerimento: utilizzate tre variabili, come segue:

`counter`:	Un contatore per contare fino a 10 (controlla quanti numeri sono stati immessi ed elaborati).
`number`:	L'ultimo numero digitato alla tastiera.
`largest`:	Il numero maggiore trovato "finora".

2.21 Scrivete un programma che utilizzi cicli e sequenze di escape di tabulazione `\t` per visualizzare la seguente tabella di valori:

```
N      10*N    100*N   1000*N

1      10      100     1000
2      20      200     2000
3      30      300     3000
4      40      400     4000
5      50      500     5000
```

2.22 Con un approccio simile a quello dell'Esercizio 2.20, trovate i due interi più grandi in una serie di 10 numeri. Nota: potete immettere un solo numero una volta.

2.23 Modificate il programma in Figura 2.11 inserendo un controllo sui valori ricevuti in input. Per ogni valore richiesto, se l'utente digita un numero diverso da 1 e da 2, non accettatelo e continuate a richiedere un valore corretto.

2.24 Che cosa visualizza questo programma?

```
#include <iostream.h>
int main()
{
   int count = 1;
   while ( count <= 10 ) {
      cout << ( count % 2 ? "*****"   "++++++++" )
           << endl;
      ++count;
   }
   return 0;
}
```

2.25 Che cosa visualizza questo programma?

```
#include <iostream.h>
int main()
{
   int row = 10, column;
   while ( row >= 1 ) {
      column = 1;
      while ( column <= 10 ) {
         cout << ( row % 2 ? "<"   ">" );
         ++column;
      }
      --row;
```

```
      cout << endl;
   }
   return 0;
}
```

2.26 (*Problema dell'*`else` *pendente*) Determinate l'output di ognuna delle seguenti costrutti, quando `x` vale `9` e `y` vale `11`, e quando `x` vale `11` e `y` vale `9`. Ricordate che il compilatore ignora gli spazi di rientro di un programma. Inoltre esso associa sempre una `else` con la `if` precedente, a meno che non sia specificato diversamente con una coppia di parentesi graffe. A prima vista però il programmatore può confondere la `if` a cui si riferisce una `else`, scivolando nel problema dello "`else` pendente" Abbiamo eliminato gli spazi di rientro nell'esercizio per renderlo più difficile. Suggerimento: applicate le convenzioni di rientro che avete imparato.

a)
```
if ( x < 10 )
   if ( y > 10 )
      cout << "******" << endl;
   else
      cout << "#####" << endl;
      cout << «SSSSS» << endl;
```

b)
```
if ( x < 10 ) {
   if ( y > 10 )
      cout << "******" << endl;
   }
   else {
      cout << "#####" << endl;
      cout << «SSSSS» << endl;
   }
```

2.27 (*Un altro problema dell'*`else` *pendente*) Modificate il codice seguente in modo da generare l'output mostrato. Utilizzate tecniche di rientro appropriate. L'unica modifica possibile sul codice è l'inserimento di parentesi graffe. Ricordate che il compilatore ignora gli spazi di rientro. Nell'esercizio abbiamo eliminato i rientri per renderlo più difficile. Nota: è anche possibile che non sia necessaria alcuna modifica.

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << «#####» << endl;
cout << «SSSSS» << endl;
cout << «&&&&&» << endl;
```

a) Ipotizzando `x = 5` e `y = 8`, viene generato il seguente output.
```
@@@@@
SSSSS
&&&&&
```

b) Ipotizzando `x = 5` e `y = 8`, viene generato il seguente output:
```
@@@@@
```

c) Ipotizzando `x = 5` e `y = 8`, viene generato il seguente output.
```
@@@@@
&&&&&
```

d) Ipotizzando `x = 5` e `y = 7`, viene generato il seguente output. Nota: le tre ultime istruzioni di output dopo la `else` fanno parte di un'istruzione composta.
```
#####
SSSSS
&&&&&
```


2.28 Scrivete un programma che legga la misura del lato di un quadrato e visualizzi un quadrato il cui lato formato del numero di asterischi digitato. Gli unici caratteri stampati sono gli asterischi e gli spazi. Il programma deve funzionare per qualsiasi numero intero tra 1 e 20. Per esempio, se l'utente digita 5, il programma visualizza
```
*****
*   *
*   *
*   *
*****
```

2.29 Un numero che si legge allo stesso modo da sinistra a destra e da destra a sinistra è detto palindromo. Per esempio, i seguenti numeri sono palindromi: 12321, 55555, 45554 e 11611. Scrivete un programma che legga un intero di cinque cifre e determini se si tratta di un palindromo. Suggerimento: utilizzate gli operatori modulo e divisione per separare i numeri nelle singole cifre che li compongono.

2.30 Richiedete in input un intero che contenga solo 0 e 1, cioè un numero "binario", e convertitelo nel suo decimale equivalente. Suggerimento: utilizzate gli operatori modulo e divisione per leggere le cifre binarie una alla volta. Nel sistema decimale la cifra più a destra ha un valore posizionale di 1, mentre le cifre più a sinistra assumono un valore posizionale multiplo di 10 (10, 100, 1000 e così via). Allo stesso modo nel sistema binario la cifra più a destra ha un valore posizionale di 1, mentre le cifre più a sinistra assumono un valore posizionale multiplo di 2 (2, 4, 8 e così via). Di conseguenza il numero decimale 234 può essere visto come 4 * 1 + 3 * 10 + 2 * 100. L'equivalente decimale del numero binario 1101 è 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 ovvero 1 + 0 + 4 + 8 cioè 13.)

2.31 Scrivete un programma che visualizzi la seguente scacchiera
```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```
utilizzando soltanto tre istruzioni di output, in una di queste forme:
```
cout << '* ';
cout << ' ';
cout << endl;
```

2.32 Scrivete un programma che visualizza i multipli del numero 2, senza mai terminare, cioè in un ciclo infinito. Che accade quando lo eseguite?

2.33 Scrivete un programma che legge il raggio di un cerchio (come valore a virgola mobile) e calcola il diametro, la circonferenza e l'area del cerchio. Per pi greco utilizzate il valore 3.14159.

2.34 Che cosa c'è di sbagliato nell'istruzione seguente? Scrivete la versione corretta, prevedendo quelle che erano probabilmente le intenzioni del programmatore.
```
cout << ++( x + y );
```

2.35 Scrivete un programma che legge tre valori a virgola mobile non nulli e determina se essi rappresentano i lati di un triangolo.

2.36 Scrivete un programma che legge tre numeri interi non nulli e determina se essi rappresentano i lati di un triangolo rettangolo.

2.37 Un azienda desidera effettuare la trasmissione dei dati sulla linea telefonica, ma teme che questi possano essere intercettati. I dati sono sequenze di interi di quattro cifre. La compagnia vi ha chiesto di scrivere un programma che crittografi i dati, in modo da poterli trasmettere con un margine di sicurezza maggiore. Il programma in questione dovrebbe leggere un intero di quattro cifre e crittografarlo con la procedura che segue. Per prima cosa va sostituita ogni cifra x con (x + 7) % 10. Successivamente vanno scambiate di posto la prima e la terza cifra del numero, così come la seconda e la quarta. A quel punto il programma è in grado di visualizzare la cifra crittografata. Scrivere anche un secondo programma che legge un intero crittografato e lo decritta, restituendo il numero originario.

2.38 *Il fattoriale di un numero non negativo n si scrive n! (e si legge "n fattoriale") ed è definito come* segue:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (per n maggiore o uguale a 1)

e

$n! = 1$ (per $n = 0$).

Per esempio $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ cioè 120.

a) Scrivete un programma che legge un intero non negativo e ne calcola il fattoriale.
b) *Scrivete un programma che stima il valore della costante matematica e utilizzando la* formula:
c) Scrivete un programma che calcola il valore di e^x utilizzando la formula

2.39 Trovate l'errore/gli errori presenti in questi segmenti di codice:

a)
```
For ( x = 100, x >= 1, x++ )
   cout << x << endl;
```
b) Il codice seguente deve visualizzare se un numero è pari o dispari:
```
switch ( value % 2 ) {
   case 0:
      cout << "Even integer" << endl;
   case 1:
      cout << "Odd integer" << endl;
}
```
c) Il codice seguente deve visualizzare gli interi dispari da 19 a 1:
```
for ( x = 19; x >= 1; x += 2 )
   cout << x << endl;
```
d) Il codice seguente deve visualizzare gli interi pari da 2 a 100:
```
counter = 2;
do {
   cout << counter << endl;
   counter += 2;
} While ( counter < 100 );
```

2.40 Scrivete un programma che somma una sequenza di numeri interi. Il primo numero letto indica il numero di valori da immettere. Il programma deve leggere solo un valore per istruzione di input. Una tipica sequenza di input può essere

```
5 100 200 300 400 500
```

dove il primo valore indica che devono essere sommati i 5 valori digitati successivamente.

2.41 Scrivete un programma che calcola e visualizza la media di un numero di interi non predefinito. Supponete che l'ultimo valore che il programma leggerà è il numero sentinella 9999. Una tipica sequenza di input può essere

```
10 8 11 7 9 9999
```

dove sarà calcolata la media di tutti i valori che precedono 9999.

2.42 Che cosa fa questo programma?

```
#include <iostream.h>
int main()
{
   int x, y;
   cout << "Enter two integers in the range 1-20: ";
   cin >> x >> y;
   for ( int i = 1; i <= y; i++ ) {
      for ( int j = 1; j <= x; j++ )
         cout << '@';
      cout << endl;
   }
   return 0;
}
```

2.43 Scrivete un programma che determini il più piccolo di una serie di numeri interi. Il primo valore letto dal programma indica il numero di valori della serie.

2.44 Scrivete un programma che calcola e visualizza il prodotto degli interi dispari da 1 a 15.

2.45 La funzione *fattoriale* viene utilizzata con una certa frequenza nei problemi statistici. Scrivete un programma che calcola i fattoriali dei numeri interi da 1 a 5. Visualizzate i risultati in formato tabulare. Quale difficoltà incontrereste se voleste calcolare il fattoriale di 20?

2.46 *Modificate il programma dell'interesse composto della Sezione 2.15 in modo tale che consideri* anche tassi di interesse del 5%, del 6%, del 7%, dell'8%, del 9% e del 10%. Utilizzate un ciclo per far variare il tasso di interesse.

2.47 Scrivete un programma che visualizza queste figure, una sotto l'altra utilizzando i cicli `for`. Tutti gli asterischi dovrebbero essere visualizzati da un'unica istruzione, del tipo `cout << '*';` (in questo modo gli asterischi saranno visualizzati affiancati). Suggerimento: le ultime due figure richiedono che le linee inizino con un numero appropriato di spazi bianchi. Versione più difficile: Combinate il codice delle quattro figure in un solo programma che le visualizzi tutte insieme, facendo un uso intelligente dei cicli `for` nidificati.

```
(A)      (B)      (C)      (D)
*        **********       **********              *
**       *********         *********             **
***      ********           ********            ***
****     *******   *******           ****
*****    ******     ******           *****
******   *****       *****           ******
*******  ****         ****           *******
********         ***           ***         ********
*********        **             **       *********
**********       *               *      **********
```

2.48 Un'applicazione informatica interessante consiste nel disegno di grafici a istogramma, o diagrammi a barre. Scrivete un programma che legge cinque numeri tra 1 e 30. Per ogni numero, il programma deve visualizzare quel numero di asterischi adiacenti. Per esempio, se l'utente digita 7, il programma visualizza *******

2.49 Un'azienda che vende per corrispondenza tratta cinque prodotti diversi, i cui prezzi al dettaglio sono i seguenti: prodotto 1 = $2.98, prodotto 2 = $4.50, prodotto 3 = $9.98, prodotto 4 = $4.49 e prodotto 5 = $6.87. Scrivete un programma che legge una serie di coppie di numeri, in questo modo:

a) Numero del prodotto
b) Quantità vendute in una giornata

Il programma dovrebbe utilizzare un'istruzione `switch` per determinare il prezzo al dettaglio di ciascun prodotto. Il suo scopo è calcolare il totale del venduto nella settimana appena trascorsa.

2.50 Modificate il programma di Figura 2.22 in modo che calcoli la media dei punti di una classe. Un giudizio 'A' vale 4 punti, 'B' vale 3 punti, e così via.

2.51 Modificate il programma in Figura 2.21 in modo che utilizzi soltanto interi per il calcolo dell'interesse composto. Suggerimento: Trattate tutti i valori monetari come un numero intero di centesimi di dollaro. Alla fine ricavate dal risultato la porzione di dollari e quella di centesimi utilizzando la divisione e il modulo. Fra le due porzioni inserite un punto decimale.

2.52 Per ipotesi supponiamo che `i = 1, j = 2, k = 3` e `m = 2`. Che cosa visualizza ciascuna delle seguenti istruzioni? Per ognuna di esse, stabilite se le parentesi sono realmente necessarie.

```
a) cout << ( i == 1 ) << endl;
b) cout << ( j == 3 ) << endl;
c) cout << ( i >= 1 && j < 4 ) << endl;
d) cout << ( m <= 99 && k < m ) << endl;
e) cout << ( j >= i || k == m ) << endl;
f) cout << ( k + m < j || 3 - j >= k ) << endl;
g) cout << ( !m ) << endl;
h) cout << ( !( j - m ) ) << endl;
i) cout << ( !( k > m ) ) << endl;
```

2.53 Scrivete un programma che visualizza una tabella che contiene la rappresentazione binaria, ottale ed esadecimale dei numeri compresi tra 1 e 256. Se non avete familiarità con le rappresentazioni dei numeri in altre basi, consultate prima l'Appendice E.

2.54 Calcolate il valore di pi greco tramite la serie infinita.
Visualizzate una tabella che mostra i valori di pi greco approssimati a 1 termine, 2 termini, tre termini della serie e così via. Quanti termini della serie dovrete prendere in considerazione per approssimare pi greco a 3.14? e a 3.141? a 3.1415? a 3.14159?

2.55 (*Terne pitagoriche*) Un triangolo rettangolo può avere per lati tutti numeri interi. I tre valori formano una cosiddetta ternaa pitagorica. I tre valori devono soddisfare la condizione che la somma dei quadrati costruiti sui due cateti deve uguagliare il quadrato costruito sull'ipotenusa. Trovate tutte le terne pitagoriche per i cateti `side1`, `side2` e per l'ipotenusa `hypotenuse`, per valori tutti minori di 500. Utilizzate un ciclo `for` a tre livelli di nidificazione, che verifica tutte le possibilità. Certo, avrete un esempio di calcolo "a forza bruta", per tentativi. Vedremo che per molti problemi non è stato ancora trovato un algoritmo risolutivo, per cui non rimane che utilizzare l'approccio a forza bruta.

2.56 Un'azienda differenzia i salari tra manager (che ricevono un fisso settimanale), lavoratori orari (che ricevono un fisso orario per le prime 40 ore settimanale, e tale fisso maggiorato del 50% per le ore di straordinario a partire dalla 41-esima), lavoratori per commissione (che ricevono $250 più il 5.7% di commissione sulle vendite all'ingrosso settimanali) e lavoratori a cottimo (che ricevono un fisso per pezzo lavorato, ogni lavoratore si dedica a un solo tipo di pezzo). Scrivete un programma che calcola la paga oraria di ciascun impiegato. Non conoscete il numero di impiegati in anticipo. Ogni tipo di impiegato ha il suo codice: i manager hanno codice 1, i lavoratori orari 2, i lavoratori per commissione 3 e i lavoratori a cottimo 4. Utilizzate un costrutto `switch` per calcolare la paga di ciascun impiegato, sulla base del codice corrispondente. Nella `switch` chiedete all'utente (cioè all'impiegato dell'ufficio stipendi) di immettere gli altri dati eventualmente necessari per il calcolo del salario, sulla base del codice digitato.

2.57 (*Leggi di De Morgan*) In questo capitolo abbiamo parlato degli operatori logici `&&`, `||` e `!`. Le leggi di De Morgan possono tornare utili per esprimere un'espressione logica nel modo più conveniente. Queste leggi dicono che l'espressione *!(condition1 && condition2)* è logicamente equivalente all'espressione *(!condition1 || !condition2)*. Inoltre l'espressione *!(condition1 || condition2)* è logicamente equivalente all'espressione *(!condition1 && !condition2)*. Utilizzate queste leggi per riformulare le espressioni seguenti, e scrivete un programma che utilizzi le due espressioni equivalenti:

```
a) !( x < 5 ) && !( y >= 7 )
b) !( a == b ) || !( g != 5 )
c) !( ( x <= 8 ) && ( y > 4 ) )
d) !( ( i > 4 ) || ( j <= 6 ) )
```

2.58 Scrivete un programma che visualizza un rombo di asterischi. Potete utilizzare istruzioni di output che visualizzano un solo asterisco o una sola spaziatura. Massimizzate l'uso dell'iterazione (con cicli `for` nidificati) e minimizzate il numero di istruzioni di output.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

2.59 Modificate il programma dell'Esercizio 2.58 in modo che richieda all'utente un numero dispari tra 1 e 19 che indichi il numero di righe del rombo.

2.60 Una delle critiche che si muovono alle istruzioni `break` e `continue` è che esse determinano delle eccezioni alle regole di strutturazione dei programmi. Possiamo affermare che queste due istruzioni possono essere sempre sostituite con dei controlli strutturati, anche se a volte non risulta evidente. Descrivete in linea generale come rimuovere tutte le `break` di un ciclo, sostituendole con dei controlli equivalenti ma strutturati. Suggerimento: l'istruzione `break` viene richiamata dall'interno di un ciclo per abbandonarlo. L'unico altro modo di uscire da un ciclo è non soddisfare più la condizione che lo controlla. Ideate un modo per inserire nella condizione del ciclo una seconda condizione, che equivalga all'uscita prematura dal ciclo di una `break`. Utilizzate la tecnica che descrivete qui per eliminare tutte le `break` dal programma in Figura 2.26.

2.61 Che cosa fa questo segmento di programma?

```
for ( i = 1; i <= 5; i++ ) {
   for ( j = 1; j <= 3; j++ ) {
      for ( k = 1; k <= 4; k++ )
         cout << '*';
      cout << endl;
   }
   cout << endl;
}
```

2.62 Descrivete a grandi linee come rimuovere le istruzioni `continue` da un ciclo, sostituendole con dei controlli equivalenti ma strutturati. Utilizzate la tecnica che descrivete per rimuovere tutte le istruzioni `continue` dal programma in Figura 2.27.

2.63 (*Testi delle canzoni*) Scrivete un programma che utilizza l'iterazione e il costrutto `switch` per visualizzare le strofe di una canzone che vi piace. Un costrutto `switch` dovrebbe visualizzare il numero della strofa, mentre un altro costrutto `switch` dovrebbe visualizzare la strofa.

L'Esercizio 2.64 si riferisce alla Sezione 2.22, "Pensare in termini di oggetti"

2.64 Descrivete in 200 parole o meno cos'è e cosa fa un'automobile. Elencate separatamente i nomi e i verbi. Nel testo abbiamo osservato come i nomi corrispondono agli oggetti di cui abbiamo bisogno per implementare il sistema, in questo caso una macchina. Prendete cinque degli oggetti che avete elencato e per ognuno di essi elencatene attributi e comportamenti. Descrivete brevemente l'interazione tra questi oggetti. Se riuscite a farlo, siete a un buon punto di una tipica progettazione orientata agli oggetti.

2.65 (*Problema di Peter Minuit*) Si tramanda che nel 1626 Peter Minuit abbia acquistato Manhattan per un valore $24.00 con un baratto. Fece un buon investimento? Per rispondere, modificate il programma dell'interesse composto in Figura 2.21, per considerare un valore iniziale di $24.00 e per calcolare l'interesse composto di quella somma, se fosse stata mantenuta in un deposito bancario fino ad oggi (372 anni in tutti fino al 1998). Eseguite il programma con i tassi di interesse di 5%, 6%, 7%, 8%, 9% e 10% e dite un po' se non siete estasiati dalle meraviglie dell'interesse composto.

CAPITOLO 3

Le funzioni

Obiettivi

- Imparare a creare programmi modulari composti da mattoni fondamentali detti funzioni
- Imparare a scrivere le proprie funzioni
- Comprendere i meccanismi alla base del passaggio di informazioni tra le funzioni
- Apprendere le tecniche di simulazione per mezzo della generazione di numeri casuali
- Comprendere il concetto di visibilità di un identificatore
- Imparare a scrivere funzioni che richiamano se stesse

3.1 Introduzione

I programmi che risolvono problemi reali sono generalmente di dimensioni molto maggiori rispetto a quelli che vi abbiamo presentato finora. L'esperienza insegna che la scelta migliore per sviluppare ed effettuare la manutenzione di grossi progetti sta nel suddividerli in componenti elementari, ognuno dei quali è una porzione del tutto più facile da gestire. Si tratta della famosa tecnica del *divide et impera*: dividi e comanda. In questo capitolo illustreremo alcune caratteristiche fondamentali del C++ che agevolano la progettazione, l'implementazione, la messa a punto e la manutenzione di programmi di notevoli dimensioni.

3.2 I componenti di un programma in C++

I componenti di un programma in C++ sono le *funzioni* e le *classi*. Un programma tipico si compone di molte funzioni e classi già fornite "di serie" con il compilatore (cioè le funzioni della libreria standard) e di funzioni e classi *definite dalle librerie di classi disponibili. In questo capitolo ci occuperemo delle funzioni e rimandiamo il discorso sulle classi al Capitolo 6.* La libreria standard del C++ offre una ricca collezione di funzioni che effettuano operazioni comuni, come i calcoli matematici, la manipolazione di stringhe e di caratteri, l'input/output e la verifica di errori. Grazie a queste funzioni, la vostra attività di programmatori si semplifica notevolmente, perché esse effettuano la maggior parte delle operazioni necessarie in un programma. Le funzioni della libreria standard sono distribuite assieme agli ambienti di sviluppo C++.

Buona abitudine 3.1

Cercate di acquisire una certa familiarità con la ricca collezione di funzioni e classi presenti nella libreria standard del C++

Ingegneria del software 3.1

Non inventate ogni volta la ruota: quando è possibile, utilizzate le funzioni della libreria standard anziché scrivere nuove funzioni. In questo modo ridurrete i tempi di sviluppo dei vostri programmi.

Obiettivo portabilità 3.1

Utilizzando le funzioni della libreria standard renderete i vostri programmi più portabili.

Obiettivo efficienza 3.1

Non cercate di riscrivere funzioni di libreria esistenti per renderle più efficienti: in genere non è possibile migliorarle rispetto a come sono già.

È possibile scrivere delle nuove funzioni che eseguono dei compiti specifici, le quali possono essere utilizzate in diversi punti del programma. Queste funzioni si chiamano *funzioni definite dal programmatore*.

Una funzione viene *invocata*, cioè viene eseguita, tramite una *chiamata di funzione*: questa specifica il nome della funzione e fornisce le informazioni (gli *argomenti*) di cui la funzione ha bisogno per effettuare i suoi compiti. Possiamo paragonare la gerarchia delle funzioni ai ruoli di un'azienda: il boss (la *funzione chiamante*) chiede a un impiegato (la *funzione chiamata*) di effettuare un certo compito e di *restituire* i risultati, a fine lavoro. La funzione boss non sa come la funzione impiegato eseguirà il compito: la funzione impiegato può anche chiedere aiuto ad altre funzioni impiegato, senza che il boss ne venga a conoscenza. Vedremo come questa possibilità di nascondere i dettagli di implementazione sia un concetto importante dell'ingegneria del software. In Figura 3.1 potete vedere la funzione `main` che comunica con diverse funzioni "impiegato" in modo gerarchico. Osservate che `worker1` è una sorta di funzione boss per `worker4` e `worker5`. Le relazioni tra funzioni possono assumere anche altre forme, non solo quella gerarchica.

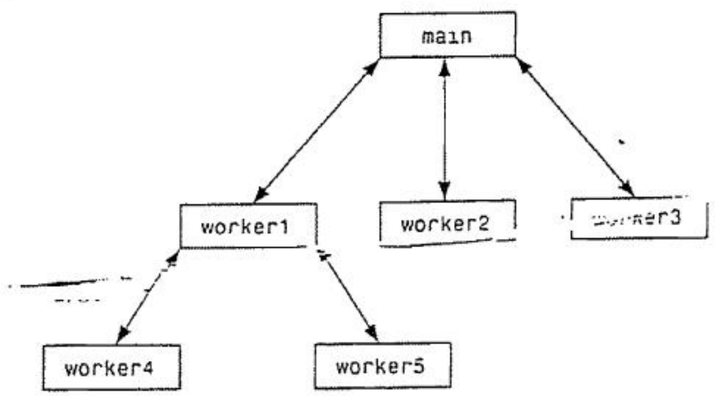


Figura 3.1 Relazione gerarchica tra la funzione boss e la funzione impiegato.

3.3 Le funzioni matematiche della libreria standard

Le funzioni matematiche di libreria eseguono i calcoli matematici più comuni. Abbiamo scelto di utilizzarle qui per introdurre i concetti fondamentali relativi alle funzioni. Nel seguito del testo mostreremo altre funzioni della libreria standard.

Normalmente per chiamare una funzione si utilizza la seguente notazione: il nome della funzione, una parentesi tonda aperta, l'*argomento* della funzione (o una lista di argomenti separati da virgole, se ve n'è più di uno) e una parentesi tonda chiusa. Per esempio per calcolare la radice quadrata di `900.0` si scrive

```
cout << sqrt(900.0);
```

Quando il computer incontra questa istruzione, chiama la funzione di libreria `sqrt` che calcola la radice quadrata del numero contenuto tra le parentesi. Il numero `900.0` è l'*argomento* di `sqrt`: questa istruzione quindi visualizza il valore `30.0`. La funzione `sqrt` prende un argomento di tipo `double` e ne restituisce uno dello stesso tipo. La restituzione di un valore `double` è una caratteristica comune delle funzioni matematiche della libreria standard. Per poter utilizzare le funzioni matematiche della libreria standard dovete includere nel programma il file di intestazione `math.h`. Nella versione più recente della libreria questo file ha cambiato nome, e ora si chiama `cmath`.

Errore tipico 3.1

Se utilizzate le funzioni matematiche ma dimenticate di includere il file di intestazione `math.h`, il compilatore vi segnalerà un errore di sintassi. Ogni funzione della libreria standard richiede l'inclusione di qualche file di intestazione all'interno del programma.

Gli argomenti di una funzione possono essere costanti, variabili o espressioni. Se `c1 = 13.0`, `d = 3.0` e `f = 4.0`, l'istruzione

```
cout << sqrt( c1 + d * f );
```

calcola e visualizza la radice quadrata di `13.0 + 3.0 * 4.0 = 25.0`, cioè 5. Scriviamo 5 anziché `5.0` perché normalmente il C++ non visualizza il punto decimale e i decimali nulli in un numero a virgola mobile. In Figura 3.2 trovate un riepilogo delle funzioni matematiche più comuni. Le variabili `x` e `y` sono di tipo `double`.

Metodo	Descrizione	Esempio
`ceil(x)`	arrotonda `x` al più piccolo intero non minore di `x`	`ceil(9.2)` dà `10.0` `ceil(-9.8)` dà `-9.0`
`cos(x)`	coseno trigonometrico di `x` (`x` è in radianti)	`cos(0.0)` dà `1.0`
`exp(x)`	funzione esponenziale di `x` (ex)	`exp(1.0)` dà `2.71828` `exp(2.0)` dà `7.38906`
`fabs(x)`	valore assoluto di `x`	se `x>0` allora `abs(x)` dà `x` se `x=0` allora `abs(x)` dà `0.0` se `x<0` allora `abs(x)` dà `-x`

Figura 3.2 Funzioni della libreria matematica di uso comune (continua)

Metodo	Descrizione	Esempio
`floor(x)`	arrotonda `x` al più grande intero non maggiore di `x`	`floor(9.2)` dà `9.0` `floor(-9.8)` dà `-10.0`
`fmod(x,y)`	resto di `x/y` (valore a virgola mobile)	`fmod(13.657,2.333)` dà `1.992`
`log(x)`	logaritmo naturale di `x` (in base `e`)	`log(2.718282)` dà `1.0` `log(7.389056)` dà `2.0`
`log10(x)`	logaritmo di `x` (in base `10`)	`log(10.0)` dà `1.0` `log(100.0)` dà `2.0`
`pow(x,y)`	`x` elevato alla `y`-esima potenza (`xy`)	`pow(2,7)` dà `128` `pow(9,.5)` dà `3`
`sin(x)`	seno trigonometrico di `x` (`x` espresso in radianti)	`sin(0.0)` dà `0`
`sqrt(x)`	radice quadrata di `x`	`sqrt(900.0)` dà `30.0` `sqrt(9.0)` dà `3.0`
`tan(x)`	tangente trigonometrica di `x` (`x` espresso in radianti)	`tan(0.0)` dà `0`

Figura 3.2 Funzioni della libreria matematica di uso comune.

3.4 Le funzioni

Le funzioni consentono ai programmatori di suddividere un programma in moduli. Le variabili dichiarate nelle definizioni delle funzioni sono dette *variabili locali*, perché sono valide soltanto all'interno della funzione in cui sono definite. La maggior parte delle funzioni prevede una lista di *parametri*, che rappresentano le informazioni che le funzioni si comunicano l'una all'altra. Anche i parametri di una funzione sono variabili locali.

Ingegneria del software 3.2

In programmi che contengono molte funzioni, l'implementazione di `main` *dovrebbe consistere semplicemente in un gruppo di chiamate ad altre funzioni che effettuano le operazioni di base del programma.*

Esistono diverse ragioni per decomporre in funzioni un programma: l'approccio *divide et impera* rende certamente più gestibile la scrittura di un programma. Un'altra ragione sta nel fatto che il software scritto in questo modo è riutilizzabile, perché le funzioni possono diventare i mattoni fondamentali di nuovi programmi. Il riutilizzo del software è uno degli obiettivi fondamentali della programmazione orientata agli oggetti. Se adottate delle buone convenzioni per i nomi e le definizioni delle funzioni, potete creare nuovi programmi utilizzando funzioni che effettuano compiti specifici, anziché scrivere ogni volta nuovo codice. Un'ultima ragione sta anche nell'evitare di ripetere continuamente delle porzioni di codice utilizzate in diverse parti di un programma: impacchettando il codice ripetuto in una funzione, lo si può eseguire ogni volta che serve tramite una semplice chiamata di funzione.

Ingegneria del software 3.3

Una funzione dovrebbe eseguire un solo compito ben definito, che si dovrebbe poter intuire dal suo nome. Funzioni di questo tipo favoriscono la scrittura di software riutilizzabile.

Ingegneria del software 3.4

Se non vi viene in mente un nome conciso che esprima precisamente il compito di una vostra funzione, è possibile che essa "faccia troppo", riunendo troppi compiti insieme: chiedetevi se non sia il caso di suddividerla in diverse funzioni più piccole.

3.5 La definizione di una funzione

Tutti i programmi che vi abbiamo presentato finora consistevano in una sola funzione di nome `main` che chiamava altre funzioni di libreria per eseguire i propri compiti. Adesso impareremo a definire nuove funzioni.

In Figura 3.3 è presentato un programma che contiene una funzione definita dall'utente, di nome `square`, per il calcolo dei quadrati dei numeri interi da 1 a 10.

```
1   // Fig. 3.3: fig03_03.cpp
2   // Creazione e uso di una funzione definita dal
3   // programmatore
4   #include <iostream.h>
5
6   int square( int );   // prototipo di funzione
7
8   int main()
9   {
10     for ( int x = 1; x <= 10; x++ )
11        cout << square( x ) << "  ";
12
13     cout << endl;
14     return 0;
15  }
16
17  // Definizione della funzione
18  int square( int y )
19  {
20     return y * y;
21  }
```

```
1  4  9  16  25  36  49  64  81  100
```

Figura 3.3 Una funzione definita dall'utente.

Buona abitudine 3.2

Lasciate delle linee vuote tra le definizioni delle diverse funzioni, in modo da tenerle separate: ciò migliora la leggibilità dei vostri programmi.

La funzione `square` viene invocata in `main` con

```
square( x )
```

La funzione `square` riceve una copia del valore di `x` nel parametro `y` e poi calcola il valore `y * y`. Il risultato viene passato nuovamente a `main`, nel punto in cui era stata invocata `square`, e `main` lo visualizza. Osservate che il valore di `x` non viene modificato dalla funzione. Questo procedimento viene ripetuto dieci volte in un ciclo `for`.

La definizione di `square` evidenzia che la funzione si aspetta un parametro intero `y`. La parola riservata `int`, che precede il nome della funzione, indica che `square` restituisce un risultato intero. L'istruzione `return` in `square` restituisce il risultato del calcolo alla funzione chiamante.

La linea 6

```
int square(int);      // prototipo di funzione
```

è un *prototipo di funzione*. Il tipo di dato `int` fra parentesi informa il compilatore che la funzione `square` si aspetta un valore di tipo `int` dalla funzione chiamante. Il tipo di dato `int` a sinistra del nome informa il compilatore che `square` restituisce alla funzione chiamante un risultato di tipo `int`. Il compilatore fa riferimento al prototipo di funzione per controllare diverse cose, e cioè che `square` restituisca il tipo di dato corretto, che prenda *un numero corretto di argomenti, che ognuno di essi sia del tipo giusto e che siano messi* nell'ordine corretto. Se la definizione della funzione compare nel programma prima dell'utilizzo effettivo della funzione, il prototipo di funzione non è necessario: in questo caso, infatti, la definizione della funzione funge anche da prototipo. Se le linee 17-20, in Figura 3.3, si trovassero prima di `main`, il prototipo di funzione della linea 5 non sarebbe necessario. Parleremo più diffusamente dei prototipi di funzione nella Sezione 3.6.

Il formato di una definizione di funzione è:

tipo-restituito nome-di-funzione (lista-di-parametri)
{
  dichiarazioni e istruzioni
}

nome-di-funzione è un qualsiasi identificatore valido mentre *tipo-restituito* è il tipo di dato che la funzione restituisce alla funzione chiamante. Se *tipo-restituito* è `void`, la funzione non restituisce alcun valore. Se *tipo-restituito* non viene specificato esplicitamente, il compilatore assume che sia `int`.

Errore tipico 3.2

Se omettete tipo-restituito *nella definizione di una funzione (e questo è diverso da* `int`*) commettete un errore di sintassi.*

Errore tipico 3.3

Se una funzione deve restituire un valore ma dimenticate di scrivere l'istruzione `return` *necessaria, commettete un errore di sintassi.*

Errore tipico 3.4

Se una funzione è dichiarata come `void`*, non può restituire alcun valore. Se ne restituite uno, commettete un errore di sintassi.*

Buona abitudine 3.3

Anche se intendete restituire un valore `int`*, non omettete* tipo-restituito*, ma scrivetelo esplicitamente anche se non è necessario.*

lista-di-parametri è la lista dei parametri ricevuti dalla funzione, separati da virgole. Se una funzione non riceve alcun valore, *lista-di-parametri* è la parola `void` o è lasciato semplicemente vuoto. È necessario elencare esplicitamente ciascun parametro.

Errore tipico 3.5

È errato dichiarare due parametri dello stesso tipo nella forma `float x, y` *anziché nella forma* `float x, float y`*. La dichiarazione* `float x, y` *contiene un errore di sintassi, perché per ogni parametro deve essere specificato il tipo di dato a cui appartiene.*

Errore tipico 3.6

Se digitate un punto e virgola dopo la parentesi destra che termina la lista dei parametri, commettete un errore di sintassi.

Errore tipico 3.7

Se definite una variabile locale che ha lo stesso nome di un parametro della funzione, commettete un errore di sintassi.

Buona abitudine 3.4

Anche se non è sbagliato, evitate di utilizzare dei nomi uguali per gli argomenti che passate a una funzione e i nomi dei parametri come compaiono nella definizione della funzione. In questo modo eviterete di confondervi.

Errore tipico 3.8

La coppia di parentesi () per la chiamata di una funzione è un operatore del C++. Il suo compito è appunto invocare la funzione specificata. Se dimenticate le parentesi in una chiamata di funzione non commettete un errore di sintassi, ma la funzione non verrà chiamata (e probabilmente non è ciò che volete).

Le *dichiarazioni* e le *istruzioni* tra le parentesi graffe formano il *corpo*, o *blocco* della funzione. Un blocco è semplicemente un'istruzione composta che può contenere delle dichiarazioni. Le variabili possono essere dichiarate in qualsiasi blocco, e i blocchi possono essere nidificati. *Non è possibile in alcun modo definire una funzione all'interno di un'altra funzione.*

Errore tipico 3.9

Se definite una funzione all'interno di un'altra funzione commettete un errore di sintassi.

Buona abitudine 3.5

Scegliendo nomi significativi per le vostre funzioni e per i loro parametri migliorerete la leggibilità dei programmi, e non sarà necessario commentarli in modo eccessivo.

Ingegneria del software 3.5

Idealmente, una funzione dovrebbe avere una lunghezza che le consenta di essere visualizzata interamente nella finestra dell'editor. In ogni caso, indipendentemente dalla sua lunghezza, essa dovrebbe effettuare un solo compito. Questo principio è alla base del riutilizzo del software.

Ingegneria del software 3.6

I programmi dovrebbero essere pensati come insiemi di funzioni di piccole dimensioni: in questo modo essi risultano essere più semplici da scrivere, collaudare e modificare.

Ingegneria del software 3.7

Se una funzione richiede un numero di parametri eccessivo, probabilmente effettua troppi compiti. Considerate l'idea di suddividerla in funzioni più piccole. Come regola pratica, non scrivete definizioni di funzione più lunghe di una riga.

Errore tipico 3.10

Se prototipo, intestazione e chiamate a una funzione non concordano per numero, tipo o ordine degli argomenti passati o per il tipo di risultato restituito, commettete un errore di sintassi.

Ci sono tre modi per restituire il controllo del programma al punto in cui una funzione è stata chiamata originariamente. Se la funzione non restituisce alcun valore, il controllo ritorna al punto originario del programma non appena la funzione termina, cioè dopo la *parentesi graffa chiusa, oppure con l'istruzione*

```
return;
```

Invece, se la funzione restituisce un risultato, l'istruzione

```
return espressione;
```

restituisce il valore di *espressione* alla funzione chiamante.

Il nostro secondo esempio utilizza la funzione definita dall'utente `maximum` per determinare il più grande di tre numeri interi (Figura 3.4). All'inizio l'utente immette tre numeri interi, poi questi sono passati alla funzione `maximum` che restituisce a `main` il maggiore, e infine questo valore viene assegnato alla variabile `largest` e inviato in output.

```
1   // Fig. 3.4: fig03_04.cpp
2   // Trova il più grande di tre interi
3   #include <iostream.h>
4
5   int maximum( int, int, int );   // prototipo di funzione
6
7   int main()
8   {
9      int a, b, c;
10
11     cout << "Enter three integers: ";
12     cin >> a >> b >> c;
```

Figura 3.4 La funzione maximum (continua)


```
13
14     // a, b e c sono gli argomenti della
15     // chiamata alla funzione maximum
16     cout << "Maximum is: " << maximum( a, b, c ) << endl;
17
18     return 0;
19  }
20
21  // Definizione della funzione maximum
22  // x, y e z sono i parametri della
23  // definizione della funzione maximum
24  int maximum( int x, int y, int z )
25  {
26     int max = x;
27
28     if ( y > max )
29        max = y;
30
31     if ( z > max )
32        max = z;
33
34     return max;
35  }
```

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 92 35 14
Maximum is: 92

Enter three integers: 45 19 98
Maximum is: 98
```

Figura 3.4 La funzione maximum.

3.6 I prototipi di funzione

Fra le caratteristiche più salienti del C++ un posto rilevante lo occupano certamente i *prototipi di funzione*. Il prototipo di una funzione serve a informare il compilatore del nome della funzione, del tipo del valore restituito e del numero, del tipo e dell'ordine dei suoi parametri. Il compilatore usa i prototipi per assicurarsi che le chiamate a quella funzione siano corrette. Le prime versioni del C++ non controllavano i prototipi, per cui era possibile chiamare una funzione in modo scorretto senza che il compilatore generasse alcun avvertimento. In fase di esecuzione, però, potevano verificarsi errori più che fatali, o in un'ipotesi migliore errori logici difficili da individuare. I prototipi di funzione furono introdotti proprio per far fronte a questa mancanza.

Ingegneria del software 3.8

In C++ i prototipi di funzione sono obbligatori. Utilizzate le direttive al preprocessore `#include` *per includere nel programma i prototipi delle funzioni della libreria standard contenuti negli opportuni file di intestazione. Se fate parte di un gruppo di lavoro, utilizzate le* `#include` *per includere i prototipi contenuti nei file di intestazione definiti dai vostri collaboratori.*

Ingegneria del software 3.9

Se una funzione compare nel programma prima del suo uso effettivo, non è necessario includerne un prototipo. In questo caso le informazioni contenute nel prototipo possono essere ricavate direttamente dalla definizione della funzione.

Il prototipo di funzione di `maximum` in Figura 3.4 è

```
int maximum(int,int,int);
```

Il prototipo indica che la funzione maximum prende tre argomenti di tipo `int`, e restituisce pure un valore di tipo `int`. Osservate che il prototipo è simile all'intestazione della definizione di `maximum`, eccetto il fatto che i nomi dei parametri (`x`, `y` e `z`) non sono esplicitati.

Buona abitudine 3.6

Molti programmatori esplicitano i nomi dei parametri anche nei prototipi di funzione, per documentarli meglio. Il compilatore ignora questi nomi.

Errore tipico 3.11

Non dimenticate il punto e virgola al termine di un prototipo di funzione, altrimenti commettete un errore di sintassi.

Il nome della funzione e il tipo di ciascun argomento formano la *segnatura* della funzione: essa non include il tipo di dato restituito.

Errore tipico 3.12

Le chiamate a una funzione devono corrispondere al suo prototipo, altrimenti si commette un errore di sintassi.

Errore tipico 3.13

La definizione di una funzione deve corrispondere al suo prototipo, altrimenti si commette un errore di sintassi.

Per fare un esempio dell'Errore tipico 3.13, se il prototipo in Fig. 3.4 fosse scritto nellaforma

```
void maximum(int,int,
```

il compilatore segnalerebbe un errore, perché il tipo `void` del prototipo è diverso dal tipo `int` presente nell'intestazione della funzione.

Un altro compito importante dei prototipi sta nella *conversione forzata degli argomenti* nel tipo di dato appropriato. Per esempio, la funzione matematica di libreria `sqrt` può essere chiamata correttamente anche con un argomento intero, sebbene in `math.h` sia specificato un argomento di tipo `double` per `sqrt`. L'istruzione


```
cout << sqrt(4);
```

calcola correttamente la radice quadrata di 4 e visualizza il valore `2`. Grazie al prototipo di funzione, il compilatore sa che deve convertire il valore intero `4` nel valore `4.0`, di tipo `double`, prima di chiamare `sqrt`. In generale, se si passano argomenti che non coincidono esattamente con il tipo dichiarato nel prototipo, questi vengono convertiti automaticamente prima che la funzione sia effettivamente chiamata. Tali conversioni, però, possono portare a risultati errati se non si seguono correttamente le *regole di promozione* del C++: queste regole indicano come convertite correttamente i valori, da un tipo a un altro, senza perdita di informazione. Nel nostro esempio, un tipo `int` è convertito in `double` senza alcuna modifica sostanziale del suo valore. Nella conversione opposta, invece, da `double` a `int`, si perde la parte decimale del numero. Anche la conversione di interi di precisione elevata in interi di precisione più piccola (per esempio da `long` a `short`) può comportare una perdita di dati.

Le regole di promozione si applicano a espressioni che contengono valori di due o più tipi di dato, che prendono il nome di *espressioni miste*. Tutti i valori di un'espressione mista sono promossi al tipo di dato più "generale" dell'espressione (conversione che non comporta perdita di informazione). In realta ciò che accade è che viene creata una copia temporanea dei valori nel tipo di dato "più generale", mentre i valori originari restano inalterati. Un altro tipo di promozione si ha quando il tipo di un argomento non corrisponde esattamente al tipo dichiarato nella definizione della funzione. In Figura 3.5 elenchiamo i tipi di dato predefiniti, dal tipo più "più generale" al tipo "meno generale"

Tipi di dato

`long double`	
`double`	
`float`	
`unsigned long int`	(sinonimo di `unsigned long`)
`long int`	(sinonimo di `long`)
`unsigned int`	(sinonimo di `unsigned`)
`int`	
`unsigned short int`	(sinonimo di `unsigned short`)
`short int`	(sinonimo di `short`)
`unsigned char`	
`short`	
`char`	

Figura 3.5 Gerarchia delle promozioni per i tipi di dati predefiniti.

Nella conversione verso i tipi "meno generali" si possono generare dei valori scorretti. Di conseguenza la conversione verso un tipo "meno generale" può avvenire solamente assegnandolo esplicitamente a una variabile di quel tipo o tramite l'operatore di cast. Un argomento di una funzione è convertito nel tipo di dato del parametro, secondo le indicazioni del prototipo, come se fosse assegnato direttamente a una variabile di quel tipo. Se la nostra `square`, che prende un parametro intero (Figura 3.3), viene chiamata con un argo-

mento reale, quest'ultimo viene convertito in `int` (un tipo "meno generale") e di conseguenza in questo caso `square` può restituire un valore non valido. Ad esempio, `square(4.5)` restituisce `16` e non `20.25`.

Errore tipico 3.14

La conversione di un dato da un tipo di dato "più generale" a un tipo "meno generale", nella gerarchia delle promozioni, può modificarne il valore.

Errore tipico 3.15

Se dimenticate il prototipo di una funzione ed essa non è definita prima del suo utilizzo effettivo, commettete un errore di sintassi.

Ingegneria del software 3.10

Un prototipo di funzione che si trova al di fuori delle definizioni di funzione si applica a tutte le chiamate che compaiono dopo il prototipo all'interno del file. Un prototipo di funzione che si trova all'interno della dichiarazione di funzione si applica soltanto alle chiamate effettuate in quella funzione.

3.7 I file di intestazione

Ogni libreria standard ha un suo *file di intestazione* corrispondente; esso contiene i prototipi di tutte le funzioni della libreria e le definizioni dei tipi e delle costanti correlati a tali funzioni. La Figura 3.6 elenca i più comuni file di intestazione della libreria standard del C++. Il termine *macro*, che troverete spesso in Figura 3.6, verrà spiegato in dettaglio nel Capitolo 6 del volume Tecniche Avanzate. I file di intestazione che terminano in `.h` seguono le convenzioni dello stile C, per ognuno di essi, esiste oggi una nuova versione. Torneremo su questo argomento quando parleremo dei nuovi standard.

File di intestazione della libreria standard	Contiene
File di intestazione "vecchio stile"	
`<assert.h>`	Macro e informazioni per la diagnostica e il debugging dei programmi. La nuova versione di questo file è `<cassert>`
`<ctype.h>`	Prototipi delle funzioni che verificano determinate proprietà dei caratteri, di funzioni che li convertono da minuscoli in maiuscoli e viceversa. La nuova versione di questo file è `<cctype>`.
`<float.h>`	Limiti dimensionali dei numeri a virgola mobile per il proprio sistema. La nuova versione di questo file è `<cfloat>`
`<limits.h>`	Limiti dimensionali degli interi per il proprio sistema. La nuova versione di questo file è `<climits>`
`<math.h>`	Prototipi delle funzioni matematiche di libreria. La nuova versione di questo file è `<cmath>`

Figura 3.6 I file di intestazione della libreria standard (continua)

File di intestazione della libreria standard	Contiene
`<stdio.h>`	Prototipi dello funzioni che effettuano le operazioni di input/output e le informazioni utilizzate da queste funzioni. La nuova versione di questo file è `<cstdio>`.
`<stdlib.h>`	Prototipi per le funzioni che convertono numeri in stringhe di caratteri, o stringhe di caratteri in numeri, per l'allocazione dinamica della memoria, la generazione di numeri casuali e di altre funzioni di utilità. La nuova versione di questo file è `<cstdlib>`
`<string.h>`	Prototipi per le funzioni di elaborazione stringhe (in stile C). La nuova versione di questo file è `<cstring>`
`<time.h>`	Prototipi di funzione e tipi per la manipolazione di date e tempi. La nuova versione di questo file è `<ctime>`
`<iostream.h>`	Prototipi delle funzioni standard per l'input/output. La nuova versione di questo file è `<iostream>`
`<iomanip.h>`	Prototipi dei manipolatori di stream per la formattazione dei dati in input/output. La nuova versione di questo file è `<iomanip>`
`<fstream.h>`	Prototipi delle funzioni per l'input/output dei file su disco (cfr. Capitolo 3 del volume Tecniche Avanzate). La nuova versione di questo file è `<fstream>`
File di intestazione "nuovo stile"	
`<utility>`	Classi e funzioni utilizzate da molti file di intestazione della libreria standard.
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	Classi contenitore della libreria standard. Le classi contenitore servono a memorizzare i dati durante l'esecuzione di un programma. Le discuteremo nel Capitolo 9 del volume Tecniche Avanzate.
`<functional>`	Classi e funzioni utilizzate dagli algoritmi della libreria standard.
`<memory>`	Classi e funzioni utilizzate dalla libreria standard per l'allocazione dinamica della memoria per le classi contenitore.
`<iterator>`	Classi per la manipolazione dei dati delle classi contenitore della libreria standard.
`<algorithm>`	Funzioni per la manipolazione delle classi contenitore della libreria standard.
`<exception>` `<stdexcept>`	Classi utilizzate per la gestione delle eccezioni (cfr. Capitolo 1 del volume Tecniche Avanzate.

Figura 3.6 I file di intestazione della libreria standard (continua)

File di intestazione della libreria standard	Contiene
`<string>`	Definizione della classe `string` della libreria standard (cfr. Capitolo 8 del volume Tecniche Avanzate).
`<sstream>`	Prototipi delle funzioni per l'input/output di stringhe in memoria (cfr. Capitolo 3 del volume Tecniche Avanzate).
`<locale>`	Classi e funzioni normalmente utilizzate nell'elaborazione degli stream, per elaborare i dati in una forma naturale per le diverse lingue e convenzioni nazionali (per es., i formati monetari, l'ordinamento di stringhe, la presentazione di caratteri ecc.).
`<limits>`	Una classe per definire i limiti dei tipi di dati numerici su ogni piattaforma hardware.
`<typeinfo>`	Classi per l'identificazione dei tipi durante l'esecuzione (determinazione dei tipi di dato durante l'esecuzione).

Figura 3.6 I file di intestazione della libreria standard.

Il programmatore può creare file di intestazione personalizzati il cui nome dovrebbe sempre terminare in `.h`. Anche per includere file di intestazione definiti dal programmatore si deve utilizzare la direttiva `#include` all'interno del programma. Per esempio, per includere il file di intestazione `square.h` dovrete scrivere

```
#include "square.h"
```

all'inizio del programma. Nella Sezione 6.2 del volume Tecniche Avanzate parliamo più in dettaglio dell'inclusione dei file di intestazione all'interno dei programmi.

3.8 La generazione di numeri casuali

Ora facciamo una breve (e speriamo gradita) digressione nelle applicazioni più popolari: vogliamo esplorare, infatti, le simulazioni e i giochi. In questa sezione e nella prossima cercheremo di sviluppare un giochino ben strutturato e suddiviso in diverse funzioni. Questo programma utilizzerà quasi tutte le strutture di controllo che abbiamo studiato finora.

Nell'aria dei casinò c'è qualcosa che tutti respirano, da coloro che puntano milioni ai tavoli da gioco più noiosi a quelli che puntano le monetine alle slot machine. È l'elemento dell'azzardo e del rischio: la possibilità di ritrovarsi miliardi in tasca dopo aver tirato fuori tutt'al più uno sgualcito biglietto da diecimila. Quest'elemento può essere introdotto anche nei programmi per mezzo della funzione `rand` facente parte della libreria standard. Prendiamo in considerazione la seguente istruzione:

```
i = rand();
```

La funzione `rand` genera un numero intero compreso tra 0 e il valore della costante simbolica `RAND_MAX`, definita nel file di intestazione `<stdlib.h>` Il valore di `RAND_MAX` deve essere almeno 32767, il massimo valore positivo che si può rappresentare con un intero di 2 byte (16 bit). Se è vero che `rand` produce numeri casuali, è anche vero che tutti i numeri

nell'intervallo hanno *uguale probabilità* di essere estratti ad ogni chiamata della funzione. Spesso l'intervallo di numeri prodotti dalla funzione `rand` non è adatto all'applicazione che si vuole scrivere. Per esempio un programma che simula il lancio di una moneta ha bisogno di due soli valori casuali (ad esempio, 0 per testa e 1 per croce) mentre un programma che simula invece il lancio di un dado ha bisogno di un intervallo che vada da 1 a 6; un gioco spaziale in cui deve apparire a caso un tipo di astronave, scelta fra quattro possibili alternative, ha bisogno dei valori da 1 a 4.

Per mostrare l'utilizzo di `rand` scriviamo un programma che simula 20 lanci consecutivi di un dado e visualizza il valore di ogni tiro. Il prototipo di `rand` si trova in `<stdlib.h>` Per ottenere il risultato voluto si utilizza l'operatore modulo (%) assieme a `rand` nel modo seguente:

```
rand() % 6
```

il che produce numeri interi compresi tra 0 e 5. In casi come questo, si suole dire che si applica un *fattore di scala*, che nel nostro caso è il numero 6. Possiamo poi *traslare* l'intervallo, portandolo da 0-5 a 1-6, aggiungendo semplicemente il valore 1. La Figura 3.7 conferma che i risultati cadono nell'intervallo da 1 a 6 come previsto. Per dimostrare che i numeri sono stati generati approssimativamente con la stessa probabilità proviamo a simulare non 20 ma 6000 lanci con il programma in Figura 3.8: in questa sequenza, ogni intero da 1 a 6 dovrebbe apparire circa 1000 volte.

Osservate che il programma non entrerà mai nella clausola `default` del costrutto `switch`; ciononostante, l'abbiamo scritta lo stesso per motivi di uniformità e per buona abitudine. Dopo aver studiato gli array, nel Capitolo 4, vedremo come sia possibile rimpiazzare l'intera `switch` con un unica istruzione.

```
1  // Fig. 3.7: fig03_07.cpp   Shifting e applicazione di
2  // un fattore di scala: uso dell'espressione 1 + rand() % 6
3  #include <iostream.h>
4  #include <iomanip.h>
5  #include <stdlib.h>
6
7  int main()
8  {
9     for ( int i = 1; i <= 20; i++ ) {
10       cout << setw( 10 ) << ( 1 + rand() % 6 );
11
12       if ( i % 5 == 0 )
13          cout << endl;
14    }
15
16    return 0;
17 }
```

```
    5         5         3         5         5
    2         4         2         5         5
    5         3         2         2         1
    5         1         4         6         4
```

Figura 3.7 Valori traslati e in scala prodotti da *1 + rand() % 6*.

```
1  // Fig. 3.8: fig03_08.cpp
2  // Lancio di un dado 6000 volte
3  #include <iostream.h>
4  #include <iomanip.h>
5  #include <stdlib.h>
6
7  int main()
8  {
9     int frequency1 = 0, frequency2 = 0,
10        frequency3 = 0, frequency4 = 0,
11        frequency5 = 0, frequency6 = 0,
12        face;
13
14    for ( int roll = 1; roll <= 6000; roll++ ) {
15       face = 1 + rand() % 6;
16
17       switch ( face ) {
18          case 1:
19             ++frequency1;
20             break;
21          case 2:
22             ++frequency2;
23             break;
24          case 3:
25             ++frequency3;
26             break;
27          case 4:
28             ++frequency4;
29             break;
30          case 5:
31             ++frequency5;
32             break;
33          case 6:
34             ++frequency6;
35             break;
36          default:
37             cout << "should never get here!";
38       }
39    }
40
41    cout << "Face" << setw( 13 ) << "Frequency"
42         << "\n    1" << setw( 13 ) << frequency1
43         << "\n    2" << setw( 13 ) << frequency2
44         << "\n    3" << setw( 13 ) << frequency3
45         << "\n    4" << setw( 13 ) << frequency4
46         << "\n    5" << setw( 13 ) << frequency5
47         << "\n    6" << setw( 13 ) << frequency6 << endl;
```

Figura 3.8 *6000 lanci di un dado a sei facce (continua)*


```
48
49       return 0;
50    }
```

```
Face      Frequency
1              987
2              984
3             1029
4              974
5             1004
6             1022
```

Figura 3.8 6000 lanci di un dado a sei facce..

Collaudo e messa a punto 3.1

Scrivete sempre una clausola `default` *in un costrutto* `switch`. *Anche se siete sicuri al 100% che il vostro programma funzioni, facendo in questo modo è possibile cautelarsi da errori non previsti.*

La seconda esecuzione del programma in Figura 3.7 produce questi risultati:

```
5     5     3     5     5
2     4     2     5     5
5     3     2     2     1
5     1     4     6     4
```

Osservate che la sequenza di valori è la stessa di prima. A questo punto avrete certamente il dubbio se questi valori siano poi così casuali. Ironicamente questa prevedibilità è una caratteristica importante della funzione `rand`. Nel debugging di un programma diviene addirittura una certezza, per verificare che le correzioni che sono apportate di volta in volta abbiano un senso.

La funzione `rand` produce in realtà *numeri pseudocasuali*; le chiamate a `rand` producono ripetutamente delle stesse sequenze di numeri, che sembrano casuali. In realtà, i numeri *sono* casuali, ma la loro sequenza si ripete uguale a se stessa ad ogni esecuzione del programma.Questa caratteristica è utilizzata per verificare gli errori dei programmi che fanno uso della funzione `rand`. Una volta terminato il debugging del programma, però, si desidererà avere una sequenza diversa in ogni esecuzione successiva. Per fare ciò, si effettua un'operazione detta di "*randomizzazione*", per mezzo della funzione di libreria `srand`. Essa prende un argomento intero `unsigned` e lo utilizza come *seme* delle sequenze casuali, consentendo alla funzione `rand` di produrre una sequenza diversa di numeri casuali ad ogni esecuzione.

L'utilizzo di `srand` è illustrato in Figura 3.9. Nel programma utilizziamo il tipo di dato `unsigned`, un'abbreviazione che sta per `unsigned int`. Un numero `int` è memorizzato in due byte di memoria e può assumere valori positivi e negativi. Una variabile di tipo `unsigned int` assume solo valori positivi, nell'intervallo da 0 a 65.535. Un `unsigned int` di quattro byte può assumere valori compresi tra 0 e 4.294.967.295. La funzione `srand` richiede un argomento di tipo `unsigned int`. Il suo prototipo è nel file di intestazione `<stdlib.h>` (`cstdlib` nel nuovo standard).

Adesso eseguiamo il programma diverse volte e osserviamone i risultati. Notate che a ogni esecuzione del programma abbiamo, stavolta, una sequenza diversa di numeri casuali.

```
1   // Fig. 3.9: fig03_09.cpp
2   // Randomizzazione del programma di lancio dei dadi
3   #include <iostream.h>
4   #include <iomanip.h>
5   #include <stdlib.h>
6
7   int main()
8   {
9      unsigned seed;
10
11     cout << "Enter seed: ";
12     cin >> seed;
13     srand( seed );
14
15     for ( int i = 1; i <= 10; i++ ) {
16        cout << setw( 10 ) << 1 + rand() % 6;
17
18        if ( i % 5 == 0 )
19           cout << endl;
20     }
21
22     return 0;
23  }
```

```
Enter seed: 67
1         6         5         1         4
5         6         3         1         2

Enter seed: 432
4         2         6         4         3
2         5         1         4         4
```

Figura 3.9 "Randomizzazione" del programma dei dadi.

Se vogliamo "randomizzare" il generatore di numeri casuali senza dover dare ogni volta un seme diverso, normalmente si utilizza un'istruzione del tipo

```
srand( time (0) );
```

Il computer leggerà l'ora corrente dall'orologio di sistema e imposterà automaticamente un nuovo valore per il seme. La funzione `time` (con l'argomento `0`) restituisce il numero di secondi trascorsi fino al momento in cui viene invocata. Questo valore viene convertito poi in un intero senza segno (`unsigned`) e utilizzato come seme per la generazione di numeri casuali. Il prototipo di funzione per `time` si trova in `<time.h>` (`ctime` nel nuovo standard).

Obiettivo efficienza 3.2

La funzione `srand` *dovrebbe essere chiamata una sola volta in un programma, per ottenere l'effetto di "randomizzazione". Più chiamate a* `srand` *sono ridondanti e riducono l'efficienza del programma.*

I valori generati direttamente da `rand` sono compresi nell'intervallo:

```
0 <= rand() <= RAND_MAX
```

In precedenza vi abbiamo fatto vedere come si può scrivere un'istruzione per simulare il lancio di un dado:

```
face = 1 + rand() % 6;
```

Questa istruzione assegna alla variabile `face` un numero intero casuale nell'intervallo da 1 a 6. Notate che l'ampiezza dell'intervallo è 6 e il numero iniziale è 1. A questo punto possiamo capire che l'ampiezza dell'intervallo è determinata dal fattore di scala utilizzato con `rand` e l'operatore modulo, mentre il numero iniziale dell'intervallo è uguale al numero sommato a `rand % 6`, cioè a 1. Generalizziamo questo concetto come segue:

```
n = a + rand() % b;
```

dove `a` è il *valore di traslazione*, cioè il primo valore dell'intervallo, e `b` è il fattore di scala, cioè il numero di interi consecutivi dell'intervallo. Negli esercizi produrremo dei numeri interi casuali che appartengono a insiemi di tutti i generi, non solo a intervalli di valori consecutivi.

Errore tipico 3.16

Utilizzare `srand` *al posto di* `rand` *per la generazione di numeri casuali è un errore di sintassi: infatti* `srand` *non restituisce alcun valore.*

3.9 I giochi d'azzardo e la parola riservata enum

Vogliamo proporvi ora uno dei giochi d'azzardo più popolari nei casinò di tutto il mondo: il *crap*; le regole del gioco sono semplici:

Un giocatore lancia due dadi a sei facce e calcola la somma dei numeri ottenuti. Se la somma vale 7 o 11 al primo lancio, il giocatore vince. Se, invece, il primo lancio dà 2, 3 o 12 il giocatore perde (e il banco vince). Se la somma del primo lancio vale 4, 5, 6, 8, 9 o 10, essa diventa "l'obiettivo" del giocatore: per vincere, il giocatore deve continuare a lanciare il dado finché non raggiunge di nuovo lo stesso valore. Il giocatore perde se ottiene un 7 prima di raggiungere l'obiettivo.

Il programma in Figura 3.10 simula questo gioco. In Figura.3.11 ne troviamo l'output.

```
1   // Fig. 3.10: fig03_10.cpp
2   // Il gioco d'azzardo crap
3   #include <iostream.h>
4   #include <stdlib.h>
5   #include <time.h>
6
7   int rollDice(void); // prototipo di funzione, lancia i dadi
```

Figura 3.10 Programma di simulazione del gioco crap (continua)

```
 8
 9  int main()
10  {
11     enum Status { CONTINUE, WON, LOST };
12     int sum, myPoint;
13     Status gameStatus;
14
15     srand( time( NULL ) );
16     sum = rollDice();              // primo lancio
17
18     switch ( sum ) {
19        case 7:
20        case 11:                    // vittoria al primo lancio
21           gameStatus = WON;
22           break;
23        case 2:
24        case 3:
25        case 12:                    // perdita al primo lancio
26           gameStatus = LOST;
27           break;
28        default:                    // ricordare l'obiettivo
29           gameStatus = CONTINUE;
30           myPoint = sum;
31           cout << "Point is " << myPoint << endl;
32           break;                  // opzionale
33     }
34
35     while ( gameStatus == CONTINUE ) {       // continua a
36                                              // lanciare i dadi
37        sum = rollDice();
38
39        if ( sum == myPoint )        // vittoria per aver
40                                     // raggiunto l'obiettivo
41           gameStatus = WON;
42        else
43           if ( sum == 7 )           // perdita per aver ottenuto 7
44              gameStatus = LOST;
45
46     }
46     if ( gameStatus == WON )
47        cout << "Player wins" << endl;
48     else
49        cout << "Player loses" << endl;
50
51     return 0;
52  }
53
```

Figura 3.10 Programma di simulazione del gioco crap (continua)


```
54  int rollDice( void )
55  {
56     int dice1, dice2, workSum;
57
58     dice1 = 1 + rand() % 6;
59     dice2 = 1 + rand() % 6;
60     workSum = dice1 + dice2;
61     cout << "Player rolled " << dice1 << " + " << dice2
62          << " = " << workSum << endl;
63
64     return workSum;
65  }
```

Figura 3.10 *Programma di simulazione del gioco crap.*

```
Player rolled 6 + 5 = 11
Player wins

Player rolled 6 + 6 = 12
Player loses

Player rolled 4 + 6 = 10
Point is 10
Player rolled 2 + 4 = 6
Player rolled 6 + 5 = 11
Player rolled 3 + 3 = 6
Player rolled 6 + 4 = 10
Player wins

Player rolled 1 + 3 = 4
Point is 4
Player rolled 1 + 4 = 5
Player rolled 5 + 4 = 9
Player rolled 4 + 6 = 10
Player rolled 6 + 3 = 9
Player rolled 1 + 2 = 3
Player rolled 5 + 2 = 7
Player loses
```

Figura 3.11 Output del gioco crap.

Osservate che il giocatore deve tirare due dadi sia la prima volta che in tutte le giocate successive. Definiamo quindi la funzione `rollDice` per tirare i dadi, calcolarne la somma e visualizzarla. La funzione `rollDice` è definita una volta sola, ma viene chiamata in due punti del programma. *Abbiamo definito `rollDice` in modo che non abbia argomenti: per* questo motivo la lista di parametri è stata indicata come `void`. La funzione `rollDice` restituisce la somma dei due dadi, per cui l'intestazione della funzione indica il tipo di dato `int`.

L'elemento aleatorio è componente fondamentale di questo gioco: il giocatore può vincere o perdere al primo tiro e in ognuno dei tiri seguenti. La variabile `gameStatus` è usata per tenere traccia della situazione ed è dichiarata come variabile di tipo `Status`. La riga

```
enum Status { CONTINUE, WON, LOST };
```

crea un *tipo di dato definito dall'utente*, che è una *enumerazione*. Un'enumerazione, introdotta dalla parola riservata `enum` e seguita dal *nome del tipo* (`Status` nel nostro caso), è composta da un insieme di costanti intere rappresentate da identificatori. La prima di queste *costanti di enumerazione* vale `0`, se non è specificato diversamente, e ogni costante successiva ha un valore incrementato di 1 rispetto a quello della precedente. Nel nostro esempio `CONTINUE` vale 0, `WON` vale 1 e `LOST` vale 2. Gli identificatori di `enum` devono essere univoci, ma più costanti di enumerazione possono assumere lo stesso valore.

Buona abitudine 3.7

Scrivete gli identificatori dei tipi definiti dall'utente con la prima lettera maiuscola.

Le variabili di tipo `Status` possono assumere soltanto uno dei tre valori dichiarati nell'enumerazione. Se il giocatore vince, `gameStatus` assume il valore `WON`, mentre se perde la variabile assume il valore `LOST`. Se il gioco è ancora in corso, `gameStatus` ha il valore `CONTINUE`, e il programma saprà che sono previsti ulteriori lanci.

Errore tipico 3.17

Se assegnate a una variabile di tipo enumerazione il numero intero corrispondente a una costante di enumerazione, commettete un errore di sintassi.

Un'altra enumerazione alquanto comune è

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT,
              NOV, DEC };
```

che crea il tipo definito dall'utente `Months` (mesi), in cui ogni costante di enumerazione rappresenta un mese dell'anno. Dato che il primo valore è impostato esplicitamente ad 1 e gli altri valori sono incrementati di 1, si crea una serie di valori compresi tra 1 e 12. Ad ogni costante di enumerazione si può assegnare un valore intero, nella definizione dell'enumerazione, e ogni costante successiva assumerà un valore incrementato di 1 unità rispetto a quello della costante precedente.

Errore tipico 3.18

Dopo aver definito una costante di enumerazione, non cercate di assegnarle un valore diverso o commetterete un errore di sintassi.

Buona abitudine 3.8

Scrivete le costanti di enumerazione con tutte le lettere maiuscole. In questo modo esse saranno più visibili nel testo del programma ed eviterete di confonderle con le variabili.

Buona abitudine 3.9

Utilizzando le enumerazioni anziché le costanti numeriche migliorerete la leggibilità dei vostri programmi.

Dopo il primo tiro, se il giocatore vince, il corpo del `while` viene saltato, perché `gameStatus` non è uguale a `CONTINUE`. Il programma continua dal costrutto `if/else` che visualizza `"Player wins"` nel caso `gameStatus` sia uguale a `WON`, o `"Player loses"` nel caso `gameStatus` sia uguale a `LOST`.

Se il gioco non finisce al primo tiro, `sum` (somma) viene salvata in `myPoint` (obiettivo del giocatore). L'esecuzione del programma continua dal costrutto `while`, perché il valore di `gameStatus` è `CONTINUE`. A ogni iterazione di `while`, viene chiamata `rollDice` per produrre una nuova `sum`. A questo punto se `sum` è uguale a `myPoint`, `gameStatus` assume il valore `WON`, la condizione di `while` non è soddisfatta, il costrutto `if/else` visualizza `"Player wins"` e l'esecuzione termina. Se `sum` è uguale a 7, `gameStatus` assume il valore `LOST`, la condizione di `while` non è soddisfatta, il costrutto `if/else` visualizza `"Player loses"` e l'esecuzione termina.

Questo programma fa un uso interessante delle varie strutture di controllo che abbiamo studiato. Le funzioni che abbiamo definito sono soltanto due: `main` e `rollDice`. Negli esercizi torneremo su questo gioco per investigarne gli aspetti più interessanti.

3.10 Le informazioni di memorizzazione

Sin dal primo capitolo abbiamo utilizzato gli identificatori dare un nome alle variabili ed in questo capitolo utilizziamo gli identificatori per dare un nome alle funzioni definite dall'utente. Un identificatore, oltre al suo nome, possiede anche altri attributi, fra cui le *informazioni di memorizzazione*, la *visibilità* o *scope* e le *informazioni di collegamento* o *linkage*.

Le *informazioni di memorizzazione* di un identificatore determinano quanto dura la sua permanenza in memoria. Alcuni identificatori esistono per brevi periodi, alcuni sono creati e distrutti ripetutamente mentre altri permangono in memoria durante l'intera esecuzione del programma.

La *visibilità* di un identificatore definisce i punti del programma in cui esso può essere riferito. Alcuni identificatori si possono utilizzare in tutto il programma, mentre altri sono validi sono in porzioni limitate del codice.

Le *informazioni di collegamento* di un identificatore determinano, nel caso di un programma composto di più file origine, se un identificatore sia noto soltanto nel file corrente o anche negli altri file. Di questo argomento parleremo più in dettaglio nel Capitolo 6.

In questa sezione parliamo degli specificatori delle informazioni di memorizzazione, mentre nella Sezione 3.11 discuteremo della visibilità degli identificatori.

Il C++ prevede quattro specificatori per le informazioni di memorizzazione di un identificatore: `auto`, `register`, `extern` e `static`.

È possibile suddividere gli identificatori in due categorie in funzione delle informazioni di memorizzazione: identificatori a *memorizzazione automatica* e identificatori a *memorizzazione statica*. Le parole riservate `auto` e `register` dichiarano variabili a memorizzazione automatica: queste sono create all'inizio del blocco in cui sono definite, esistono durante l'esecuzione di quel blocco e sono distrutte all'uscita dal blocco. Solo le variabili possono appartenere alla categoria di memorizzazione automatica. Le variabili locali e i parametri di una funzione appartengono normalmente a questa categoria. Lo

specificatore `auto` dichiara esplicitamente che una variabile appartiene alla categoria di memorizzazione automatica. Per esempio, la dichiarazione seguente indica che le variabili `x` e `y`, di tipo `float`, sono variabili locali automatiche, cioè esistono soltanto durante l'esecuzione del corpo della funzione in cui sono definite:

```
auto float x, y;
```

Le variabili locali sono automatiche per default, per cui la parola riservata `auto` si utilizza raramente. Nel corso del testo continueremo a indicare le variabili appartenenti alla categoria di memorizzazione automatica semplicemente come variabili automatiche.

Obiettivo efficienza 3.3

La memorizzazione automatica è un mezzo per risparmiare l'utilizzo della memoria, perché le variabili automatiche sono create all'inizio del blocco in cui sono dichiarate e sono distrutte all'uscita di tale blocco.

Ingegneria del software 3.11

La categoria di memorizzazione automatica incorpora il principio del minor privilegio. Perché mai una variabile dovrebbe restare in memoria quando non serve più?

Nella versione in linguaggio macchina dei programmi, i dati sono normalmente caricati in registri della CPU per essere elaborati successivamente.

Obiettivo efficienza 3.4

Scrivendo lo specificatore `register` prima della dichiarazione di una variabile automatica si suggerisce al compilatore di mantenere il più possibile la variabile nei registri ad alta velocità della CPU, anziché in memoria. Le variabili utilizzate con una certa intensità, come contatori e totali, sono delle ottime candidate ad essere contenute nei registri: in questo modo la CPU risparmia il tempo necessario per leggere ripetutamente il loro valore dalla memoria centrale.

Errore tipico 3.19

Se utilizzate più di uno specificatore di categoria di memorizzazione per un identificatore, commettete un errore di sintassi. Se indicate, per esempio, lo specificatore `register`, non potete utilizzare anche lo specificatore `auto`.

Il compilatore è libero di ignorare gli specificatori `register`. Questo può accadere, per esempio, quando non c'è un numero sufficiente di registri per tutte le variabili dichiarate come `register`. Ecco perché abbiamo detto che una dichiarazione come quella che segue semplicemente *suggerisce* che `counter` sia posto in uno dei registri della CPU:

```
register int counter = 1;
```

L'uso della parola riservata `register` è consentita soltanto per le variabili locali e i parametri delle funzioni.

Obiettivo efficienza 3.5

In alcuni sistemi non è necessario utilizzare gli specificatori `register`. I compilatori più recenti sono in grado di riconoscere automaticamente le variabili più utilizzate in determinati segmenti di codice e porle nei registri, anche se il programma non fa menzione esplicita dello specificatore `register`.

Le parole riservate `extern` e `static` si utilizzano per dichiarare identificatori di variabili e funzioni a memorizzazione statica. Variabili di questo genere esistono sin dall'inizio del programma. Per le variabili, lo spazio in memoria è allocato e inizializzato una sola volta, all'inizio dell'esecuzione del programma; per le funzioni, il nome della funzione esiste sin dall'inizio. Ad ogni modo, anche se i nomi delle variabili e delle funzioni esistono dall'inizio dell'esecuzione, non è detto che i loro identificatori possano essere utilizzati in tutto il programma. Infatti, le informazioni di memorizzazione e la visibilità degli identificatori sono due cose distinte, come vedremo più in dettaglio nella Sezione 3.11.

Ci sono due tipi di identificatori statici: gli identificatori esterni, come le variabili globali e i nomi di funzione, e le variabili locali dichiarate con lo specificatore `static`. Le variabili globali e i nomi di funzione per default sono invece `extern`. Una variabile globale si crea scrivendo la sua dichiarazione al di fuori del corpo di qualsiasi funzione e mantiene il suo valore durante l'intera esecuzione del programma. Le variabili globali e le funzioni possono essere riferite (richiamate) da qualsiasi funzione che si trovi dopo la loro dichiarazione, all'interno del file del programma.

Ingegneria del software 3.12

Se dichiarate una variabile come globale anziché locale, prestate attenzione agli effetti collaterali di questa dichiarazione: se una funzione modifica accidentalmente il valore della variabile, l'errore si propagherà a tutte le funzioni che la utilizzeranno successivamente. In generale è bene evitare l'utilizzo di variabili globali, a meno che non siano necessarie per incrementare le prestazioni di un programma.

Ingegneria del software 3.13

Le variabili utilizzate solo nel corpo di una funzione dovrebbero essere variabili locali di quella funzione, anziché globali.

Anche le variabili locali dichiarate come `static` sono note soltanto nella funzione in cui sono dichiarate, ma a differenza di quelle automatiche conservano il loro valore anche quando la funzione termina la sua esecuzione. Alla chiamata successiva di tale funzione, una variabile `static` ha ancora il valore che conteneva al termine dell'ultima esecuzione della funzione.

L'istruzione seguente dichiara la variabile locale `count` come `static` e la inizializza a 1.

```
static int count = 1;
```

Tutte le variabili di tipo numerico dichiarate come `static` sono inizializzate a zero, se il programmatore non fornisce esplicitamente un valore. Gli specificatori `extern` e `static` assumono un significato speciale quando sono utilizzati esplicitamente per identificatori esterni. Nel Capitolo 7 del volume Tecniche Avanzate ne parleremo in dettaglio, quando discuteremo dei programmi composti da più file.

3.11 Le regole di visibilità

La porzione di programma in cui esiste un identificatore prende il nome di *visibilità* o *scope*. Per esempio, se dichiariamo una variabile locale in un blocco, possiamo riferirci ad essa soltanto in tale blocco, o nei blocchi nidificati all'interno di esso. Esistono quattro tipi di visibilità degli identificatori: *a livello di funzione*, *di file*, *di blocco* e *di prototipo di funzione*. In seguito vedremo un quinto tipo di scope: *a livello di classe*.

Un identificatore dichiarato al di fuori di qualsiasi funzione ha *visibilità a livello di file*. Un identificatore di questo genere è noto a tutte le funzioni che si trovano dopo la sua dichiarazione, fino alla fine del file. Le variabili globali, le definizioni delle funzioni e i prototipi di funzione che si trovano al di fuori delle funzioni hanno tutti visibilità a livello di file.

Le *etichette*, o *label*, sono identificatori seguiti da un segno di due punti come `start:`, esse rappresentano gli unici identificatori che hanno *visibilità a livello di funzione*. Le etichette possono essere utilizzate ovunque nella funzione in cui appaiono, ma non possono essere riferite al di fuori del corpo di tale funzione. Le etichette sono utilizzate nei costrutti `switch` (nelle clausole `case`) e nelle istruzioni `goto` (lo vedremo nel Capitolo 7 del volume Tecniche Avanzate). Le funzioni *nascondono* le proprie etichette alle altre funzioni: questo *occultamento delle informazioni* è uno dei principi fondamentali nella metodologia per lo sviluppo di software di buona qualità.

Gli identificatori dichiarati in un blocco hanno *visibilità a livello di blocco*. In questo caso lo scope inizia dalla dichiarazione dell'identificatore e termina con la parentesi graffa destra (}) che segnala la fine del blocco. Le variabili locali dichiarate all'inizio di una funzione hanno visibilità a livello di blocco, così come i parametri delle funzioni, che sono assimilati alle variabili locali. Le variabili possono essere dichiarate in qualsiasi blocco. Nel caso di blocchi nidificati, se un identificatore del blocco esterno ha lo stesso nome di uno del blocco interno, l'identificatore del blocco esterno è occultato da quello del blocco interno finché il blocco interno non termina. Il blocco interno *vede* solo il proprio identificatore locale e non quello più esterno con lo stesso nome. Le variabili locali `static` hanno scope a livello di blocco anche se vengono create sin dall'inizio del programma. In questo caso, come vedete, visibilità e informazioni di memorizzazione non coincidono.

Gli unici identificatori che hanno *visibilità a livello di prototipo di funzione* sono i parametri elencati nei prototipi di funzione. Come abbiamo già accennato, non è necessario utilizzare dei nomi di identificatori nei prototipi dei nomi, ma bastano solo i tipi dei dati. Se sono presenti, il compilatore li ignorerà in ogni caso. Per questo motivo potete utilizzare i nomi che avete scritto nei prototipi di funzione in qualsiasi altro punto del programma, senza temere problemi di ambiguità.

Errore tipico 3.20

Se utilizzate accidentalmente nomi uguali per gli identificatori di due blocchi nidificati, commettete normalmente un errore logico, perché il blocco interno sarà in grado di vedere soltanto il proprio identificatore, e non potrà accedere a quello del blocco esterno.

Buona abitudine 3.10

Evitate di utilizzare nomi di variabile che occultano le variabili dei livelli di visibilità superiori. Una soluzione semplice (anche se non sempre perseguibile) è quella di non duplicare i nomi degli identificatori in tutto il programma.

Il programma in Figura 3.12 illustra le problematiche relative alla visibilità, alle variabili locali automatiche e alle variabili locali statiche.


```
1  // Fig. 3.12: fig03_12.cpp
2  // Esempio di utilizzo di livelli di visibilità diversi
3  #include <iostream.h>
4
5  void a( void );    // prototipo di funzione
6  void b( void );    // prototipo di funzione
7  void c( void );    // prototipo di funzione
8
9  int x = 1;         // variabile globale
10
11 int main()
12 {
13    int x = 5,    // variabile locale di main
14
15    cout << "local x in outer scope of main is " << x << endl;
16
17    {             // inizia un nuovo livello di visibilità
18       int x = 7;
19
20       cout << "local x in inner scope of main is " << x << endl;
21    }             // finisce il nuovo livello di visibilità
22
23    cout << "local x in outer scope of main is " << x << endl;
24
25    a();          // a ha la variabile locale automatica x
26    b();          // b ha la variabile locale statica x
27    c();          // c utilizza la variabile globale x
28    a();          // a reinizializza la var. locale automatica x
29    b();    // la v. loc. stat. x conserva il suo valore precedente
30    c();    // anche la variabile globale x conserva il suo valore
31
32    cout << "local x in main is " << x << endl;
33
34    return 0;
35 }
36
37 void a( void )
38 {
39    int x = 25;  // inizializzata ad ogni chiamata di a
40
41    cout << endl << "local x in a is " << x
42         << " after entering a" << endl;
43    ++x;
44    cout << "local x in a is " << x
45         << " before exiting a" << endl;
46 }
47
```

Figura 3.12 Esempio dei differenti tipi di scope (continua)

```
48  void b( void )
49  {
50      static int x = 50;  // Inizializzazione statica soltanto
51                          // alla prima chiamata di b.
52      cout << endl << "local static x is " << x
53           << " on entering b" << endl;
54      ++x;
55      cout << "local static x is " << x
56           << " on exiting b" << endl;
57  }
58
59  void c( void )
60  {
61     cout << endl << "global x is " << x
62          << " on entering c" << endl;
63     x *= 10;
64     cout << "global x is " << x << " on exiting c" << endl;
65  }
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 50 on entering b
local static x is 51 on exiting b

global x is 1 on entering c
global x is 10 on exiting c

local x in a is 25 after entering a
local x in a is 26 before exiting a

local static x is 51 on entering b
local static x is 52 on exiting b

global x is 10 on entering c
global x is 100 on exiting c
local x in main is 5
```

Figura 3.12 Esempio dei differenti tipi di scope.

La variabile globale x viene dichiarata e inizializzata a 1. Questa variabile risulterà occultata in tutti i blocchi e le funzioni che dichiarano una propria variabile x al loro interno. In main viene dichiarata una nuova variabile x, inizializzata a 5. main la visualizza, per mostrare che la variabile x globale risulta occultata. Successivamente, viene definito in main un nuovo blocco con un'altra variabile locale x inizializzata a 7. L'istruzione di visualizzazione consecutiva mostra come questa x occulti la x del blocco più esterno di main. La variabile x che vale 7 viene distrutta automaticamente all'uscita del blocco, e viene visualizzata la x del blocco esterno di main, che non risulta più occultata. Nel programma sono definite tre funzioni, ognuna delle quali non prende argomenti e non restituisce alcun valore. La funzione a definisce la variabile automatica x e la inizializza a 25. Quando viene

chiamata a, la variabile viene visualizzata, incrementata e visualizzata di nuovo prima dell'uscita dalla funzione. A ogni chiamata di a, la variabile automatica viene ricreata e inizializzata a 25. La funzione b dichiara la variabile statica x e la inizializza a 50. Le variabili locali dichiarate come static conservano il loro valore anche quando si esce dal loro scope. Quando viene chiamata b, x viene visualizzata, incrementata e visualizzata di nuovo prima dell'uscita dalla funzione. Nella chiamata successiva a b, la variabile statica x contiene ancora il valore 51. La funzione c non dichiara alcuna variabile propria. Di conseguenza, quando si riferisce la variabile x, viene utilizzata la x globale. Quando viene chiamata c, la variabile globale viene visualizzata, moltiplicata per 10 e visualizzata di nuovo prima dell'uscita dalla funzione. All'esecuzione successiva di c, la variabile globale conterrà il valore modificato, 10. Infine il programma visualizza la variabile x di main, per mostrare come nessuna delle funzioni abbia modificato questa x, perché tutte le altre funzioni utilizzano variabili di altri livelli di visibilità.

3.12 Il concetto di ricorsione

I programmi che abbiamo analizzato finora sono composti generalmente da funzioni che si richiamano l'un l'altra in modo disciplinato e gerarchico. Per risolvere alcuni problemi può essere utile, invece, che una funzione possa richiamare se stessa. Una *funzione ricorsiva* è una funzione che direttamente o indirettamente (tramite un'altra funzione) richiama se stessa. La ricorsione è un argomento molto importante, di cui si occupano in dettaglio i corsi universitari di informatica. In questa sezione e nella prossima presenteremo dei semplici esempi di ricorsione. La Figura 3.17 (alla fine della Sezione 3.14) riepiloga gli esempi e gli esercizi del testo che fanno uso di questo concetto.

Cerchiamo di comprendere prima come funziona la ricorsione dal punto di vista concettuale ed in seguito ne daremo degli esempi per mezzo di alcuni programmi. Tutte le soluzioni che si basano sulla ricorsione hanno dei punti in comune. In primo luogo, ovviamente, una funzione ricorsiva viene chiamata per risolvere un problema; la funzione deve saper risolvere soltanto una piccola parte del problema, cioè il caso o i casi più semplici, che prendono il nome di *casi base*. Se la funzione viene chiamata per un caso base, essa restituisce un risultato e termina; se la funzione, invece, viene chiamata per un caso più complesso, essa suddivide il problema in due parti concettualmente distinte: una prima parte che la funzione sa risolvere e una seconda che non sa risolvere. Perché il problema possa avere una soluzione ricorsiva, la seconda parte deve assomigliare al problema originario ed essere di dimensioni un po' più piccole di esso. Dato che quest'ultima parte assomiglia al problema originario, la funzione può chiamare una nuova copia di se stessa, che lavorerà sul problema leggermente più piccolo, e così via. Questa chiamata è detta anche o *passo ricorsivo*. Un passo ricorsivo prevede anche una return, in modo che il risultato sia combinato con la parte del problema che la funzione sa come risolvere: il risultato prenderà forma poco a poco e verrà restituito alla funzione chiamante, per esempio a main.

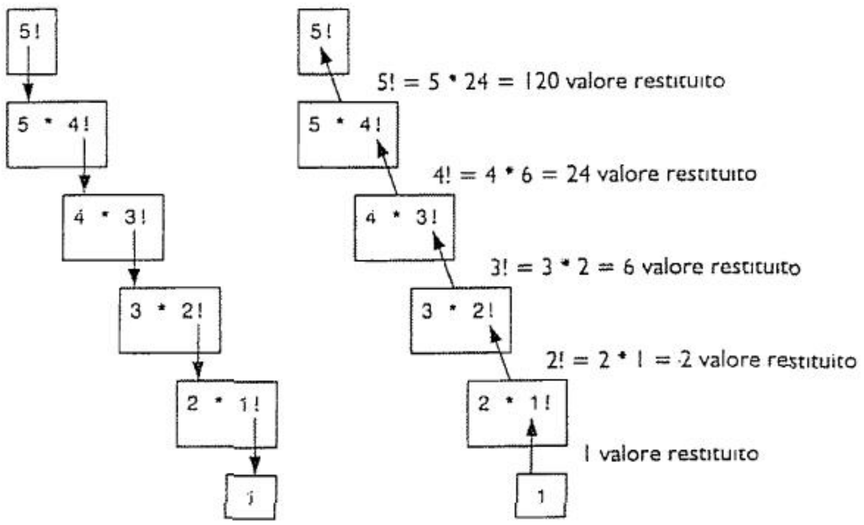
Un passo ricorsivo è eseguito mentre la chiamata originaria alla funzione è ancora pendente, ovvero non ha terminato la sua esecuzione. Il passo ricorsivo può consistere di tante chiamate alla stessa funzione, man mano che il problema continua a essere suddiviso in porzioni sempre più piccole. A un certo punto la ricorsione deve (o dovrebbe) terminare, per cui man mano che le porzioni diventano sempre più piccole, esse devono converge-

re verso il caso base. A quel punto, la funzione riconosce il caso base e lo risolve, quindi restituisce il risultato alla copia precedente della funzione; quest'ultima la restituisce alla copia immediatamente precedente e così via, lungo tutta la serie di copie fino alla chiamata originaria, che restituisce il risultato a `main`. Siamo certi che il procedimento vi sembrerà un po' esotico, per non dire cervellotico, rispetto alle tecniche di risoluzione dei problemi che abbiamo illustrato finora. È il caso quindi di passare a un esempio: scriviamo un programma ricorsivo che effettui uno dei calcoli matematici più conosciuti dagli studenti.

Il fattoriale di un numero non negativo *n*, che si scrive *n!* e si pronuncia *n fattoriale* è il prodotto

n · (*n* - 1) · (*n* - 2) · … · 1

dove 1! è uguale a 1 e, per definizione, 0! è pure uguale a 1. Per esempio 5! è il prodotto dei primi cinque numeri naturali ed è uguale a 120.



a) successione delle chiamate ricorsive b) valori restituiti a ogni chiamata ricorsiva

Figura 3.13 Calcolo ricorsivo di 5!.

Il fattoriale di un numero intero `number`, maggiore o uguale a 0, può essere calcolato con un'iterazione, cioè non ricorsivamente, con un ciclo `for`, come segue:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

Si può giungere facilmente ad una definizione ricorsiva del problema osservando che

$n! = n \cdot (n\text{-}1) \cdot (n\text{-}2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 = n \cdot [(n\text{-}1) \cdot (n\text{-}2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1] = n \cdot (n\text{-}1)!$

Ad esempio, nel caso del calcolo di 5! possiamo ridurci a calcolare 5 * 4!, poi a calcolare 5 * 4 * 3! e così via fino ai casi di base di cui sappiamo calcolare direttamente il valore (cioè 1! = 0! = 1).

Il calcolo di 5! procederebbe come in Figura 3.13. La Figura 3.13a mostra come la successione di chiamate ricorsive va avanti finché non viene calcolato 1! che vale 1: questo è l'evento che termina la ricorsione. La Figura 3.13b mostra i valori restituiti da ciascuna chiamata ricorsiva alla copia chiamante, fino al calcolo del valore finale.

Il programma in Figura 3.14 utilizza la ricorsione per calcolare e visualizzare i fattoriali dei numeri interi compresi tra 0 e 10.

```
1   // Fig. 3.14: fig03_14.cpp
2   // Funzione fattoriale ricorsiva
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   unsigned long factorial( unsigned long );
7
8   int main()
9   {
10     for ( int i = 0; i <= 10; i++ )
11        cout << setw( 2 ) << i << "! = " << factorial( i ) << endl;
12
13     return 0;
14  }
15
16  // Definizione ricorsiva della funzione factorial
17  unsigned long factorial( unsigned long number )
18  {
19     if ( number <= 1 )  // caso base
20        return 1;
21     else                // caso ricorsivo
22        return number * factorial( number - 1 );
23  }
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Figura 3.14 Calcolo dei fattoriali con una funzione ricorsiva.

La funzione ricorsiva `factorial` verifica come prima cosa se la condizione per terminare la ricorsione è vera, cioè se il numero dato è minore o uguale a 1. Se `number` è effettivamente minore o uguale a 1, `factorial` restituisce il valore 1 e termina, perché non sono necessarie altre chiamate a se stessa. Se `number` è maggiore di 1, l'istruzione

```
return number * factorial( number - 1 );
```

riformula il problema nel prodotto di `number` e del risultato della funzione `factorial` per il fattoriale di `number - 1`. Come notate, `factorial( number - 1 )` è la porzione di problema più piccola del problema originario `factorial( number )`.

La funzione `factorial` prende un parametro di tipo `unsigned long` e restituisce un risultato dello stesso tipo. Questa è la notazione abbreviata per `unsigned long int`. Una variabile di questo tipo, secondo le specifiche del linguaggio C++, occupa almeno 4 byte (32 bit) di memoria, per cui può rappresentare almeno i valori compresi nell'intervallo tra 0 e 4294967295. Come vediamo in Figura 3.14 i fattoriali raggiungono facilmente grossi valori. Abbiamo scelto il tipo `unsigned long` in modo tale che, anche sui computer che hanno interi piccoli (a 2 byte), sia possibile calcolare i fattoriali di numeri maggiori di 7. Purtroppo la funzione `factorial` produce grossi valori molto presto e neanche il tipo `unsigned long` è di grande aiuto in questo caso.

Negli esercizi esploreremo l'uso dei tipi `float` e `double` per il calcolo dei fattoriali e dei grandi numeri in genere. La limitazione di dimensione dei dati è una debolezza di molti linguaggi di programmazione: essi non sono in grado di adattarsi volta per volta alle caratteristiche uniche di un'applicazione. Tuttavia il C++ è un linguaggio estensibile, in cui è possibile creare grandi interi delle dimensioni desiderate.

Errore tipico 3.21

Se in una funzione ricorsiva dimenticate di scrivere l'istruzione che restituisce un valore, qualora sia previsto nell'intestazione della funzione, otterrete un messaggio di avvertimento del compilatore.

Errore tipico 3.22

Se omettete il caso base o scrivete in modo errato il passo ricorsivo, in modo tale che non converga mai al caso base, vi ritroverete con un problema di ricorsione infinita e, alla fine, con un errore di memoria non sufficiente. Si tratta di un problema analogo a quello dei cicli infiniti. Fate attenzione, perché la ricorsione infinita può anche essere causata da un valore ricevuto in input che la funzione non si aspettava e non sa come trattare.

3.13 Un altro esempio di ricorsione: la serie di Fibonacci

La serie di Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21....

inizia con 0 e 1 e ha la proprietà che ciascun numero della serie è la somma dei due numeri precedenti.

La serie contraddistingue diversi fenomeni naturali, in particolare essa descrive la forma di una spirale. Il rapporto dei numeri di Fibonacci successivi converge verso il valore 1,618... (cioè (1+sqrt(5))/2). Anche questo valore caratterizza diversi fenomeni naturali ed è chiamato *numero aureo* o *rapporto aureo*. È una peculiarità degli esseri umani quella di cercare valori di questo tipo, supponiamo anche per esigenze di tipo estetico oltre che scientifico.

Ci sono architetti che progettano finestre, stanze ed edifici basandosi su multipli del numero aureo; esistono persino cartoline dai lati che misurano multipli del numero aureo.

La serie di Fibonacci può essere definita ricorsivamente in questo modo:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Il programma in Figura 3.15 calcola l'*i*-esimo numero di Fibonacci utilizzando ricorsivamente la funzione `fibonacci`. Anche i numeri di Fibonacci, come i numeri fattoriali, tendono a raggiungere presto valori molto grandi. Per questo motivo abbiamo scelto anche in questo caso il tipo di dato `unsigned long` per il parametro e il tipo di dato restituito da `fibonacci`. In Figura 3.15 ogni coppia di linee di output mostra un'esecuzione del programma.

```
1   // Fig. 3.15: fig03_15.cpp
2   // Funzione di fibonacci ricorsiva
3   #include <iostream.h>
4
5   long fibonacci( long );
6
7   int main()
8   {
9      long result, number;
10
11     cout << "Enter an integer: ";
12     cin >> number;
13     result = fibonacci( number );
14     cout << "Fibonacci(" << number << ") = " << result << endl;
15     return 0;
16  }
17
18  // Definizione ricorsiva della funzione fibonacci
19  long fibonacci( long n )
20  {
21     if ( n == 0 || n == 1 )  // caso base
22        return n;
23     else                     // caso ricorsivo
24        return fibonacci( n - 1 ) + fibonacci( n - 2 );
25  }
```

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1
```

Figura 3.15 Generazione ricorsiva dei numeri di Fibonacci (continua)

```
Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Figura 3.15 Generazione ricorsiva dei numeri di Fibonacci.

La chiamata a `fibonacci` da `main` non è ricorsiva, mentre lo sono tutte le chiamate successive dall'interno della stessa funzione `fibonacci`. Ogni volta che `fibonacci` viene invocata, verifica per prima cosa se sta trattando il caso base, cioè se `n` è 0 o 1. Se è così, viene semplicemente restituito il valore `n`. Se `n` è maggiore di 1, il passo ricorsivo genera *due* chiamate ricorsive, ognuna delle quali riduce il problema a dimensioni leggermente inferiori della chiamata originaria. La Figura 3.16 illustra l'esecuzione di `fibonacci( 3 )`. In figura e nel testo abbreviamo `fibonacci` in `f` per semplicità.

La figura solleva alcune questioni interessanti sul modo in cui i compilatori prendono in considerazione i diversi operandi. Si tratta, in realtà, di un problema diverso dal modo in cui gli operatori sono applicati ai loro operandi e cioè non ha a che fare con le regole di precedenza. Dalla Figura 3.16 è chiaro che durante il calcolo di `f(3)` saranno effettuate due chiamate ricorsive a `f(2)` e `f(1)`. Ma qual è l'ordine seguito?

Molti programmatori fanno sempre l'assunzione che gli operatori siano valutati da sinistra a destra. Strano a dirsi, il C++ non specifica in alcun modo l'ordine di calcolo di molti operatori (incluso +). Per questo motivo non date per scontato l'ordine di esecuzione che vi sembra più ragionevole: le chiamate potrebbero essere effettuate nell'ordine `f(2)` e `f(1)`, o nell'ordine inverso. Nella maggior parte dei programmi, incluso il nostro, questo non influenza il risultato, ma ci sono programmi in cui il calcolo di un operando può comportare degli *effetti collaterali* sull'altro, influenzando il risultato finale dell'operazione.

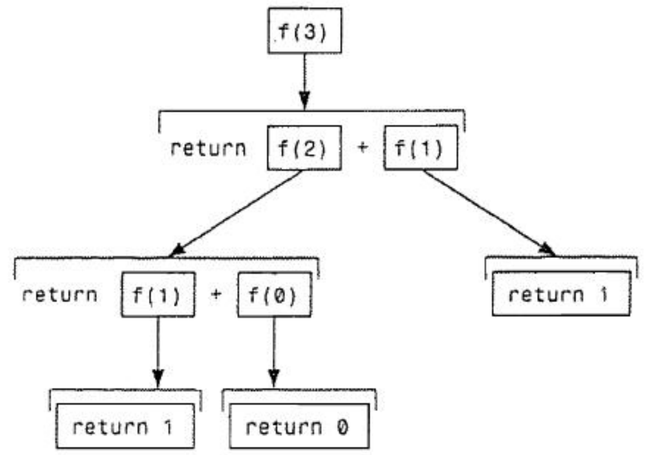


Figura 3.16 Insieme di chiamate ricorsive a fibonacci.

Il C++ specifica l'ordine di calcolo degli operandi di 4 operatori soltanto, cioè `&&`, `||`, la virgola (`,`) e `?:`. I primi tre sono operatori binari, e si garantisce che essi siano applicati agli operandi da sinistra a destra. L'ultimo e l'unico operatore ternario del C++; il suo calcolo procede nel modo seguente: innanzitutto viene valutato il primo operando, se esso non ha valore zero viene valutato il secondo operando e l'ultimo viene ignorato, altrimenti il secondo viene ignorato e viene valutato il terzo.

Errore tipico 3.23

Se scrivete del codice che dipende dall'ordine di calcolo degli operandi di operatori diversi da `&&`, `||`, `?:` e dall'operatore virgola (`,`) potreste incorrere in errori, perché i compilatori non valutano gli operandi necessariamente in un ordine prestabilito.

Obiettivo portabilità 3.2

Programmi che dipendono dall'ordine di calcolo degli operandi di operatori diversi da `&&`, `||`, `?:` e dall'operatore virgola (`,`) possono funzionare in modo diverso su sistemi diversi, e su compilatori diversi.

Vogliamo darvi qualche avvertimento sui programmi ricorsivi, come quello dei numeri di Fibonacci. Ciascun livello di ricorsione della funzione `fibonacci` ha un effetto di duplicazione sul numero di chiamate da eseguire: il numero di chiamate necessarie per calcolare l'*n*-esimo numero di Fibonacci è la *n*-esima potenza di 2. Come capite, si sfondano rapidamente i limiti accettabili. Il calcolo del 20-esimo numero di Fibonacci richiede all'incirca un milione di chiamate, e già sul 30-esimo numero arriviamo al miliardo di chiamate! Gli informatici parlano in questi casi di *complessità esponenziale*. Problemi di calcolo di questo genere, a prima vista semplici, sono in grado di umiliare anche i supercomputer più potenti! I corsi universitari di informatica trattano in modo specifico le problematiche poste dalla complessità.

Obiettivo efficienza 3.6

Evitate di risolvere problemi come il calcolo dei numeri di Fibonacci con la ricorsione, per non incorrere in un numero potenzialmente esplosivo di chiamate ricorsive.

3.14 Ricorsione o iterazione?

Nelle sezioni precedenti vi sarete fatti un'idea di come sia possibile risolvere uno stesso *problema in modo ricorsivo o iterativo. Questa sezione è stata pensata appositamente per* mettere a confronto i due approcci, e capire quando scegliere l'uno o l'altro a nelle diverse situazioni.

Ricorsione e iterazione si basano entrambe su una struttura di controllo: l'iterazione utilizza una struttura iterativa, la ricorsione una di selezione. Anche la ripetizione è alla base di entrambe, nell'iterazione addirittura in modo esplicito. Entrambe hanno poi bisogno di una condizione che le faccia terminare: l'iterazione termina quando la condizione del ciclo non è più soddisfatta, mentre la ricorsione termina quando si raggiunge il caso base. L'iterazione controllata da contatore e la ricorsione sono simili per il fatto che si avvicinano gradualmente al termine: l'iterazione modifica continuamente il contatore finché questo a un certo punto non soddisfa più la condizione del ciclo, mentre la ricorsione produce continuamente delle versioni più semplici del problema originario, fino a raggiungere la versione più semplice possibile, cioè il caso base. Sia l'iterazione che la ricorsione possono essere infinite. Per l'iterazione ciò si verifica quando la condizione non diventa mai falsa, mentre per la ricorsione quando non c'è modo di convergere verso il caso base.

La ricorsione ha molti lati negativi: il numero di chiamate alla funzione è generalmente alto e ciò rappresenta un costo in termini di tempo di elaborazione e di memoria utilizzata. Ciascuna chiamata ricorsiva, infatti, causa la creazione di un'altra copia della funzione (in realtà soltanto delle sue variabili locali) e ciò può comportare il consumo di parecchia memoria. L'iterazione invece è limitata in una funzione, per cui non si ha né un eccessivo tempo di elaborazione né un consumo eccessivo di memoria. A questo punto ci si può chiedere i motivi per cui scegliere la ricorsione come metodo per risolvere i problemi.

Ingegneria del software 3.14

Qualsiasi problema che abbia una soluzione ricorsiva, ne ha anche una iterativa. Normalmente si preferisce utilizzare quella ricorsiva quando la ricorsione rispecchia in modo più naturale il modo in cui risolvere il problema, per cui il codice risultante è più semplice da scrivere e da verificare. Un altro buon motivo per preferire la soluzione ricorsiva è l'incapacità di riuscire a elaborare una soluzione iterativa.

Obiettivo efficienza 3.7

Evitate di utilizzare la ricorsione in situazioni in cui sono cruciali le prestazioni. La ricorsione ha un costo in termini di efficienza di esecuzione e di memoria utilizzata.

Errore tipico 3.24

Se una funzione chiama accidentalmente se stessa in modo diretto o indiretto (tramite un'altra funzione), ma non è ricorsiva, commettete un errore logico.

Molti libri di testo introducono la ricorsione molto più tardi del nostro. La nostra opinione è che la ricorsione è un argomento così ricco e complesso che è meglio introdurlo brevemente subito per darne diversi esempi nel corso del testo. La Figura 3.17 riepiloga gli esempi e gli esercizi sulla ricorsione presenti in questo libro.

Capitolo	Esempi ed esercizi
Capitolo 3	Funzione fattoriale
	Funzione di Fibonacci
	Massimo comun divisore
	Somma di due interi
	Moltiplicazione di due interi
	Potenza intera di un intero
	Torre di Hanoi
	Visualizzazione inversa di input da tastiera
	Visualizzazione della ricorsione
Capitolo 4	Somma degli elementi di un array
	Visualizzazione di un array
	Visualizzazione inversa di un array
	Visualizzazione inversa di una stringa
	Controllo su una stringa per verificare se è palindroma
	Valore minimo in un array.
	Ordinamento con l'algoritmo "selectionsort"
	Problema delle otto regine
	Ricerca lineare
	Ricerca binaria
Capitolo 5	Ordinamento con l'algoritmo "quicksort"
	Attraversamento di un labirinto
	Visualizzazione inversa di una stringa digitata alla tastiera
Capitolo 4 (Tecniche Avanzate)	Inserimento di un elemento in una lista collegata
	Eliminazione di un elemento da una lista collegata
	Ricerca di un elemento in una lista collegata
	Visualizzazione inversa di una lista collegata
	Inserimento di un elemento in un albero binario
	Attraversamento di un albero binario in pre-ordine
	Attraversamento di un albero binario in in-ordine
	Attraversamento di un albero binario in post-ordine

Figura 3.17 Riepilogo degli esempi e degli esercizi del testo che utilizzano la ricorsione.

Vogliamo ora ricordarvi alcune osservazioni che vi proponiamo frequentemente nel corso del libro. Una buona progettazione è di importanza cruciale per produrre del buon software e per ottenere buone prestazioni. Una buona progettazione è la chiave per produrre software complesso e di grandi dimensioni. Le prestazioni sono di importanza vitale, perché le esigenze dei sistemi del futuro saranno sempre maggiori. In tutto ciò, che ruolo assume la scrittura di funzioni?

Ingegneria del software 3.15

La cosiddetta decomposizione in funzioni *di un programma, cioè la sua suddivisione in compiti minori svolti da funzioni, è parte integrante di una progettazione chiara e gerarchica. Ma ricordate che essa ha un costo.*

Obiettivo efficienza 3.8

Le chiamate di funzione hanno un costo in termini di tempo di elaborazione e di memoria utilizzata: un programma decomposto in modo eccessivo, in confronto ad un programma monolitico privo di funzioni, può effettuare un numero eccessivo di chiamate di funzione e, dunque, essere meno efficiente. Tuttavia i programmi monolitici sono difficili da scrivere, verificare, mantenere e aggiornare.

Quello che vogliamo dirvi è: decomponete in funzioni i vostri programmi in modo giudizioso, tenendo sempre a mente il bilancio tra prestazioni e buona progettazione.

3.15 Le funzioni che hanno una lista di parametri vuota

In C++ una lista di parametri vuota si specifica scrivendo la parola riservata `void` o non scrivendo nulla tra la coppia di parentesi tonde. La dichiarazione

```
void print();
```

indica che la funzione `print` non prende alcun argomento e non restituisce alcun valore. La Figura 3.18 mostra i due modi per dichiarare e utilizzare le funzioni che non prendono argomenti.

Buona abitudine 3.11

Definite sempre i prototipi per le vostre funzioni, anche quando non sono necessari (per esempio se le funzioni sono dichiarate prima di essere effettivamente utilizzate). Scrivendo i prototipi, infatti, potete cambiare in qualsiasi momento l'ordine delle funzioni all'interno del codice (cosa molto comune man mano che il codice evolve) senza incorrere in errori.

```
1  // Fig. 3.18: fig03_18.cpp
2  // Funzione che non prende argomenti
3  #include <iostream.h>
4
5  void function1();
6  void function2( void );
7
8  int main()
9  {
10    function1();
11    function2();
12
13    return 0;
14 }
15
16 void function1()
17 {
18    cout << "function1 takes no arguments" << endl;
19 }
```

Figura 3.18 Due modi di dichiarare e utilizzare le funzioni che non prendono argomenti (continua)


```
20
21 void function2( void )
22 {
23    cout << "function2 also takes no arguments" << endl;
24 }
```

```
function1 takes no arguments
function2 also takes no arguments
```

Figura 3.18 Due modi di dichiarare e utilizzare le funzioni che non prendono argomenti.

Obiettivo portabilità 3.3

Il significato di una lista di parametri vuota in C++ è decisamente diverso da quello che ha in C. In C una lista vuota significa che il compilatore non verifica gli argomenti passati alla funzione: il programmatore, cioè, può passarne quanti ne vuole. In C++, invece, significa che la funzione non prende argomenti. Fate attenzione, dunque, perché alcuni programmi in C possono generare errori di compilazione quando sono compilati da un compilatore C++.

Nella nostra discussione non ci stiamo curando di includere tutti i particolari: infatti una funzione definita in un file prima di una sua effettiva chiamata non ha bisogno di un prototipo separato, perché è la sua intestazione stessa a fungere da prototipo.

Errore tipico 3.25

Un programma in C++ non può essere compilato se non è presente un prototipo per ciascuna funzione o ciascuna di esse non è definita prima di essere effettivamente utilizzata.

3.16 Le funzioni in linea

È senz'altro positivo dal punto di vista della progettazione poter strutturare un programma come insieme di funzioni ma, come osservato, ogni chiamata di funzione richiede un suo tempo di elaborazione. Per questo motivo diverse funzioni, specialmente quelle più piccole, possono essere scritte come *funzioni in linea*. Il qualificatore `inline`, scritto prima del tipo di dato restituito da una funzione, suggerisce al compilatore di generare una copia del codice della funzione nei diversi punti del programma in cui essa è richiamata (e dove ciò è appropriato), anziché effettuare una vera chiamata di funzione. Invece di avere una singola copia della funzione da chiamare ogni volta che serve, il compilatore dunque inserisce nel programma il codice di tale funzione in ogni punto in cui viene invocata. Un effetto negativo è che il codice eseguibile può facilmente aumentare di dimensioni. Generalmente il compilatore ignora i qualificatori `inline`, se non per le funzioni di dimensioni più piccole.

Ingegneria del software 3.16

Se effettuate una modifica in una funzione `inline`, tutte le sue copie inserite nel programma devono essere aggiornate. Ciò può essere un problema significativo nello sviluppo e nella manutenzione di grossi progetti.

Buona abitudine 3.12

Il qualificatore `inline` *dovrebbe essere utilizzato solo per funzioni piccole e utilizzate molto frequentemente.*

Obiettivo efficienza 3.9

L'utilizzo delle funzioni `inline` *può ridurre il tempo di esecuzione, ma aumenta in genere le dimensioni del codice eseguibile.*

Il programma in Figura 3.19 utilizza la funzione in linea `cube` per calcolare il volume di un cubo di lato `s`. La parola riservata `const` nella lista dei parametri di `cube` informa il compilatore che la funzione non modifica la variabile `s`. Ciò assicura che il valore di `s` non sia modificato prima del calcolo.

Parleremo più diffusamente della parola riservata `const` nei Capitoli 4, 5 e 7.

Ingegneria del software 3.17

Molti programmatori non si prendono la briga di dichiarare i parametri di valore come `const`*, anche se la funzione chiamata non deve in alcun modo modificare l'argomento passato. Il qualificatore* `const` *protegge soltanto la copia dell'argomento originario ma non quest'ultimo.*

```
1   // Fig. 3.19: fig03_19.cpp
2   // Uso di una funzione inline per calcolare
3   // il volume di un cubo.
4   #include <iostream.h>
5
6   inline float cube( const float s ) { return s * s * s; }
7
8   int main()
9   {
10     cout << "Enter the side length of your cube:  ";
11
12     float side;
13
14     cin >> side;
15     cout << "Volume of cube with side "
16          << side << " is " << cube( side ) << endl;
17
18     return 0;
19  }
```

```
Enter the side length of your cube:  3.5
Volume of cube with side 3.5 is 42.875
```

Figura 3.19 Funzione inline che calcola il volume di un cubo.

3.17 I riferimenti e il passaggio di parametri per riferimento

In diversi linguaggi di programmazione esistono due modi per invocare una funzione e passarle dei parametri: *per valore* e *per riferimento* (o *per indirizzo*). Se un argomento viene passato per valore, ne viene effettuata preventivamente una *copia* ed è questa copia che viene passata alla funzione. In questo modo, tutte le modifiche che la funzione effettua sulla copia non influenzano il valore della variabile originaria della funzione chiamante. Questo meccanismo aiuta a prevenire eventuali *effetti collaterali*, che di frequente compromettono la correttezza dei programmi. In tutte le funzioni prese in considerazione finora abbiamo utilizzato solamente il passaggio dei parametri per valore.

Obiettivo efficienza 3.10

Uno svantaggio della chiamata per valore consiste nel fatto che la preventiva copia dei dati in memoria può incidere notevolmente sui tempi di elaborazione. Ciò è particolarmente vero nel caso di dati di grosse dimensioni.

In questa sezione introduciamo anche i cosiddetti *parametri di riferimento* (o più semplicemente *riferimenti*), che costituiscono il primo dei due modi per effettuare una chiamata per riferimento. La chiamata per riferimento consente alla funzione chiamata di accedere direttamente ai dati della funzione chiamante e, se è il caso, anche di modificarli.

Obiettivo efficienza 3.11

La chiamata per riferimento migliora le prestazioni di un programma, perché elimina il tempo di elaborazione necessario per la copia dei dati elimina il problema della copia dei dati tipico delle funzioni con parametri passati per valore e rende i programmi più efficienti.

Ingegneria del software 3.18

Una chiamata per riferimento può modificare e introdurre degli errori nei dati della funzione chiamante, per cui pone problemi di sicurezza.

Vedremo fra poco come sia possibile ottenere il vantaggio della chiamata per riferimento proteggendo allo stesso tempo i dati della funzione chiamante.

Un riferimento è un *alias* dell'argomento a cui corrisponde. Per indicare che un parametro è passato per riferimento occorre scrivere nel prototipo della funzione il segno `&` dopo il tipo di dato; ovviamente, per mantenere la consistenza delle dichiarazioni, è necessario utilizzare la stessa convenzione anche nella lista dei parametri dell'intestazione della funzione. Per esempio

```
int &count
```

nell'intestazione di una funzione dichiara `count` come un riferimento ad un argomento `int`. Quando chiamate la funzione in questione basterà menzionare il nome della variabile ed essa sarà passata per riferimento. Nel momento in cui la funzione chiamata utilizzerà la variabile `count`, essa farà riferimento alla variabile originaria della funzione chiamante e potrà leggerla e modificarla direttamente.

In Figura 3.20 mettiamo a confronto la chiamata per valore con quella per riferimento. Lo stile degli argomenti nelle chiamate a `squareByValue` e `squareByReference` è identico, cioè entrambe le variabili vengono menzionate con il loro nome. Se non esaminassimo i prototipi o le intestazioni delle funzioni, non potremmo sapere se la funzione può modificare i suoi argomenti oppure no.

```
1   // Fig. 3.20: fig03_20.cpp
2   // Confronto tra chiamata per valore e
3   // chiamata per riferimento.
4   #include <iostream.h>
5
6   int squareByValue( int );
7   void squareByReference( int & );
8
9   int main()
10  {
11     int x = 2, z = 4;
12
13     cout << "x = " << x << " before squareByValue\n"
14          << "Value returned by squareByValue: "
15          << squareByValue( x ) << endl
16          << "x = " << x << " after squareByValue\n" << endl;
17
18     cout << "z = " << z << " before squareByReference" << endl;
19     squareByReference( z );
20     cout << "z = " << z << " after squareByReference" << endl;
21
22     return 0;
23  }
24
25  int squareByValue( int a )
26  {
27     return a *= a;    // l'argomento del chiamante non è modificato
28  }
29
30  void squareByReference( int &cRef )
31  {
32     cRef *= cRef;     // l'argomento del chiamante è modificato
33  }
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Figura 3.20 Esempio di chiamata per riferimento.

Errore tipico 3.26

Dato che i riferimenti sono menzionati solo con il nome nel corpo della funzione chiamata, potreste trattare inavvertitamente un riferimento come parametro passato per valore. Ciò può causare degli effetti collaterali inattesi se la funzione chiamata modifica le variabili originali della funzione chiamante.

Nel Capitolo 5 parleremo dei puntatori: essi costituiscono la forma alternativa per effettuare una chiamata per riferimento. Nel caso dei puntatori,lo stile della chiamata sarà indicativo del fatto che si tratta una chiamata per riferimento (viene passato esplicitamente l'indirizzo di memoria di una variabile), e quindi il programmatore saprà già nel momento in cui sta scrivendo il codice che la funzione chiamata può modificare gli argomenti che le si passano.

Obiettivo efficienza 3.12

Se dovete passare un grosso tipo di dato, vi conviene utilizzare il passaggio per riferimento costante per simulare l'aspetto e la protezione di una chiamata per valore ed eliminare il tempo di elaborazione necessario per la copia del dato.

Per specificare un riferimento a un valore costante, scrivete il qualificatore `const` prima del tipo specificato nella dichiarazione del parametro.

Notate la presenza del simbolo & nella lista di parametri della funzione `squareByReference`. Alcuni programmatori preferiscono la forma `int& cref` alla forma `int &cref`.

Ingegneria del software 3.19

Per una maggiore chiarezza e per migliorare le prestazioni, molti programmatori preferiscono passare gli argomenti modificabili tramite i puntatori, gli argomenti non modificabili di piccole dimensioni tramite le chiamate per valore e quelli di grandi dimensioni con i parametri per riferimento costanti.

I riferimenti possono essere utilizzate anche come alias per altre variabili all'interno di una funzione. Per esempio il codice

```
int count = 1;              // dichiara la variabile intera count
int &cRef = count;          // crea cRef come alias di count
++cRef;                     // incrementa count (utilizzando l'alias)
```

incrementa la variabile `count` tramite il suo alias `cRef`. Le variabili che contengono riferimenti devono essere inizializzate durante la loro dichiarazione (cfr. Figura 3.21 e Figura 3.22) e non possono essere riassegnate come alias di altre variabili. Una volta dichiarata una variabile come alias di un'altra, tutte le operazioni effettuate sull'alias, cioè sul riferimento, sono effettuate sulla variabile originaria. L'alias infatti è semplicemente un nome alternativo per una variabile che già esiste. Se si prende l'indirizzo di un alias o si confrontano due alias non si ha un errore di sintassi: ciò che accade semplicemente è che qualsiasi operazione che coinvolge l'alias viene effettuata sulla variabile originaria.

Un argomento passato per riferimento deve essere un *lvalue*: non può essere una costante o un'espressione che restituisce un *rvalue*.

Errore tipico 3.27

Se dichiarate più riferimenti in un'istruzione di dichiarazione scrivendo una sola volta il simbolo & all'inizio della riga commettete un errore logico. Per dichiarare `x`, `y` *e* `z` *come tre riferimenti dovete usare la notazione* `int &x = a, &y = b, &z = c` *e non la notazione scorretta* `int& x = a, y = b, z = c;` *o* `int &x, y, z;`.

Le funzioni possono restituire riferimenti, ma l'operazione comporta dei pericoli. Se restituite un riferimento ad una variabile dichiarata nella funzione chiamata, la variabile deve essere dichiarata nella funzione come `static`. In caso contrario, restituirete il riferimento ad una variabile automatica che non esiste più dopo l'esecuzione della funzione. Ciò corrisponde a restituire una variabile "indefinita" e, dunque, il comportamento del programma che la utilizza sarà imprevedibile.

Alcuni compilatori sono in grado di verificare tale tipo di errori segnalandovi un avvertimento. I riferimenti a variabili indefinite sono detti *riferimenti pendenti*.

Errore tipico 3.28

Dovete obbligatoriamente inizializzare una variabile riferimento durante la sua dichiarazione, altrimenti commettete un errore di sintassi.

```
1   // References must be initialized
2   #include <iostream.h>
3
4   int main()
5   {
6      int x = 3, &y = x;  // y e ora un alias di x
7
8      cout << "x = " << x << endl << "y = " << y << endl;
9      y = 7;
10     cout << "x = " << x << endl << "y = " << y << endl;
11
12     return 0;
13  }
```

```
x = 3
y = 3
x = 7
y = 7
```

Figura 3.21 Utilizzo di un riferimento inizializzato.

```
1   // References must be initialized
2   #include <iostream.h>
3
4   int main()
5   {
6      int x = 3, &y;    // Errore: y deve essere inizializzato
7
```

Figura 3.22 Tentativo di utilizzare un riferimento non inizializzato (continua)


```
8      cout << "x = " << x << endl << "y = " << y << endl;
9      y = 7;
10     cout << "x = " << x << endl << "y = " << y << endl;
11
12     return 0;
13  }
```

```
Compiling FIG03_21.CPP:
Error FIG03_21.CPP 6: Reference variable 'y' must be
   initialized
```

Figura 3.22 Tentativo di utilizzare un riferimento non inizializzato.

Errore tipico 3.29

Se tentate di modificare un riferimento dichiarato precedentemente perché diventi alias di un'altra variabile commettete un errore logico. Il valore dell'altra variabile verrà semplicemente scritto nella locazione della variabile riferita originariamente dal riferimento.

Errore tipico 3.30

Se restituite un puntatore o un riferimento a una variabile automatica della funzione chiamata commettete un errore logico. Alcuni compilatori vi avvertiranno dell'operazione ambigua con un messaggio.

3.18 Gli argomenti di default

Alcune funzioni sono invocate ripetutamente con lo stesso valore per un dato argomento. Il programmatore può allora indicarlo come *argomento di default*, fornendone contestualmente un valore di default. Se in seguito si invoca la funzione ma si omette l'argomento di default, il suo valore verrà inserito automaticamente dal compilatore e la funzione lo riceverà ugualmente.

Gli argomenti di default devono trovarsi agli ultimi posti nella lista dei parametri. Se ci sono due o più argomenti di default, e ne omettete uno che non è l'ultimo nella lista dei parametri, dovete necessariamente omettere tutti gli altri che seguono. Gli argomenti di default dovrebbero essere specificati nella prima occorrenza del nome della funzione (quindi, normalmente, nel prototipo). I valori degli argomenti di default possono essere costanti, variabili globali o chiamate di funzioni. È possibile utilizzare gli argomenti di default anche con le funzioni `inline`.

La Figura 3.23 mostra l'uso degli argomenti di default nel calcolo del volume di una scatola (un parallelepipedo rettangolo). Il prototipo della funzione `boxVolume` alla linea 5 *specifica che i tre argomenti valgono 1 per default. Osservate inoltre che abbiamo scritto* i nomi delle variabili anche nel prototipo, per una migliore leggibilità. Come già osservato, ciò non è strettamente necessario.

```
1   // Fig. 3.23: fig03_23.cpp
2   // Utilizzo degli argomenti di default
3   #include <iostream.h>
4
5   int boxVolume( int length = 1, int width = 1, int height = 1 );
6
7   int main()
8   {
9      cout << "The default box volume is: " << boxVolume()
10          << "\n\nThe volume of a box with length 10,\n"
11          << "width 1 and height 1 is: " << boxVolume( 10 )
12          << "\n\nThe volume of a box with length 10,\n"
13          << "width 5 and height 1 is: " << boxVolume( 10, 5 )
14          << "\n\nThe volume of a box with length 10,\n"
15          << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
16          << endl;
17
18     return 0;
19  }
20
21  // Calcola il volume di una scatola
22  int boxVolume( int length, int width, int height )
23  {
24     return length * width * height;
25  }
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Figura 3.23 Utilizzo di argomenti di default.

La prima chiamata alla funzione `boxVolume` (linea 9) non indica argomenti, per cui utilizza i tre valori di default. La seconda chiamata (linea 11) passa l'argomento `length`, e quindi utilizza i default solo per `width` e `height`. La terza chiamata (linea 13) passa gli argomenti `length` e `width`, e quindi utilizza il valore di default solo per `height`. L'ultima chiamata (linea 15) specifica invece tutti gli argomenti, e quindi non utilizza alcun valore di default.

Buona abitudine 3.13

Gli argomenti di default semplificano la scrittura delle chiamate alle funzioni. Ad ogni modo molti programmatori avvertono la necessità di scrivere esplicitamente i valori passati, per maggiore chiarezza.

Errore tipico 3.31

Se specificate e utilizzate un argomento di default che non è in coda alla lista dei parametri, non utilizzando i default per tutti gli argomenti in coda, commettete un errore di sintassi.

3.19 L'operatore unario di risoluzione dello scope

Come sapete già, in C++ è possibile dichiarare variabili locali e globali che hanno lo stesso nome. È possibile utilizzare l'operatore unario di risoluzione dello scope (`::`) per accedere a una variabile globale all'interno di un blocco che dichiara una variabile locale con lo stesso nome. Non è possibile utilizzare questo operatore, però, per accedere a una variabile locale appartenente a un blocco esterno. Se non ci sono variabili locali che hanno il suo stesso nome, una variabile globale è accessibile anche senza l'operatore di risoluzione dello scope. Nel Capitolo 6 parleremo dell'operatore binario di risoluzione dello scope, che si utilizza con le classi.

La Figura 3.24 illustra l'uso dell'operatore unario di risoluzione dello scope con variabili locali e globali dal nome identico. Per sottolineare il fatto che le versioni globali e locali di `PI` sono due cose distinte, ne dichiariamo una come `float` e l'altra come `double`.

Errore tipico 3.32

Se cercate di accedere a una variabile non globale di un blocco esterno tramite l'operatore unario di risoluzione dello scope, commettete un errore di sintassi, se non esiste una variabile globale con quel nome. Commettete, invece, un errore logico se ne esiste una, perché ovviamente non è a tale variabile che volevate accedere.

```
1   // Fig. 3.24: fig03_24.cpp
2   // Utilizzo dell'operatore unario di risoluzione dello scope
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   const double PI = 3.14159265358979;
7
8   int main()
9   {
10     const float PI = static_cast< float >(   :PI );
11
12     cout << setprecision( 20 )
13          << "  Local float value of PI = " << PI
14          << "\nGlobal double value of PI = " << ::PI << endl;
15
16     return 0;
17  }
```

```
  Local float value of PI = 3.14159
Global double value of PI = 3.14159265358979
```

Figura 3.24 Utilizzo dell'operatore unario di risoluzione dello scope.

Buona abitudine 3.14

Evitate di riutilizzare lo stesso nome per diverse variabili a livelli di visibilità diversi in uno stesso programma. Anche se è perfettamente lecito può generare confusione.

3.20 L'overloading delle funzioni

In C++ è possibile definire diverse funzioni con lo stesso nome, purché ognuna di essa abbia un insieme di parametri diverso (almeno soltanto nei tipi dei dati, cioè nella segnatura della funzione). Questa caratteristica prende il nome di *overloading* o *sovraccarico* di una funzione. Quando si chiama una funzione sovraccarica, il compilatore sceglie la corretta funzione da chiamare esaminando il numero, il tipo e l'ordine dei parametri presenti nella chiamata. Spesso l'overloading si utilizza per scrivere diverse funzioni che effettuano più o meno le stesse operazioni, ma su tipi di dato diversi.

Buona abitudine 3.15

Effettuare l'overloading di funzioni che effettuano delle operazioni molto simili può rendere i programmi più leggibili e comprensibili.

Il programma in Figura 3.25 effettua l'overloading della funzione `square` per calcolare il quadrato di un numero `int` e di un numero `double`. Nel Capitolo 8 parleremo dell'overloading degli operatori, per definire il modo in cui devono operare su oggetti appartenenti a tipi di dato definiti dall'utente. Pensandoci attentamente, abbiamo già utilizzato degli operatori in overloading: l'operatore di inserimento nello stream << e di estrazione dallo stream >> La Sezione 3.21 introduce il concetto di template di una funzione, che serve a generare automaticamente gli overloading di una stessa funzione per effettuare le stesse operazioni su tipi di dato diversi. Parleremo più diffusamente dei template di classe e di funzione nel Capitolo 1 del volume Tecniche Avanzate.

```
1   // Fig. 3.25: fig03_25.cpp
2   // Utilizzo delle funzioni sovraccariche
3   #include <iostream.h>
4
5   int square( int x ) { return x * x; }
6
7   double square( double y ) { return y * y; }
8
9   int main()
10  {
11     cout << "The square of integer 7 is " << square( 7 )
12          << "\nThe square of double 7.5 is " << square( 7.5 )
13          << endl;
14
15     return 0;
16  }
```

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Figura 3.25 Utilizzo di funzioni sovraccariche.

Le funzioni sovraccariche si distinguono dalle loro *segnatura*: la segnatura è la combinazione del nome e dei tipi dei suoi parametri. Il compilatore codifica un identificatore di funzione insieme con il numero e il tipo dei suoi parametri, per essere certo di effettuare un collegamento coerente dal punto di vista del tipo dei dati. Il compilatore deve assicurare che ogni chiamata invochi la giusta versione della funzione, con gli argomenti conformi ai parametri. Il compilatore rileva eventuali errori in fase di collegamento e avverte il programmatore. Il programma in Figura 3.26 è stato compilato con il Borland C++. I nomi che iniziano per `@` corrispondono ai nomi delle funzioni codificati dal compilatore. La lista di questi nomi inizia con `$q`. Nella lista di parametri della funzione `nothing2`, `zc` rappresenta un `char`, `i` rappresenta un `int`, `pf` rappresenta un `float *` e `pd` rappresenta un `double *` Nella lista dei parametri della funzione `nothing1`, `i` rappresenta un `int`, `f` rappresenta un `float`, `zc` rappresenta un `char` e `pi` rappresenta un `int *`. Le due funzioni `square` si distinguono grazie alle rispettive liste di parametri: una specifica `d` come `double` e l'altra specifica `i` come `int`. Il tipo di dato restituito non viene codificato insieme alla segnatura dal compilatore. Le funzioni sovraccariche possono restituire dati uguali o differenti, ma le liste dei parametri devono essere necessariamente differenti.

Errore tipico 3.33

Se due funzioni sovraccariche hanno una lista di parametri identica si otterrà un errore di sintassi.

```
1   // Codifica dei nomi
2   int square(int x) { return x * x; }
3
4   double square(double y) { return y * y; }
5
6   void nothing1(int a, float b, char c, int *d)
7      { }  // corpo della funzione vuoto
8
9   char *nothing2(char a, int b, float *c, double *d)
10     { return 0; }
11
12  int main()
13  {
14     return 0;
15  }
```

```
public _main
public @nothing2$qsifzcpi
public @nothing1$qifzcpi
public @square$qd
public @square$qi
```

Figura 3.26 Codifica dei nomi per evitare ambiguità nella fase di linking.

L'unica cosa che un compilatore esamina per distinguere due funzioni con nome uguale è la lista dei parametri; le funzioni sovraccariche possono anche avere un numero di parametri diverso. Occorre fare una certa attenzione nell'utilizzare i parametri di default con le funzioni sovraccariche, perché ciò può generare ambiguità.

Errore tipico 3.34

Se una funzione con i valori di default omessi può essere chiamata in modo identico a un'altra funzione sovraccarica commettete un errore di sintassi. Per esempio se in un programma avete una funzione che non prende esplicitamente nessun argomento e una funzione con lo stesso nome che ha un argomento di default, commettete un errore di sintassi se cercate di effettuare una chiamata senza argomenti.

3.21 Le funzioni generiche

L'overloading serve generalmente a creare diverse funzioni che effettuano operazioni molto simili con logiche differenti o su tipi di dati diversi. Se invece logica e operazioni sono uguali per tutti i tipi di dato, potete utilizzare una caratteristica del linguaggio più conveniente e compatta, i *template di funzione* o *funzioni generiche*. Tutto quello che dovete fare è scrivere la definizione del template di una sola funzione. Sulla base del tipo di argomenti con cui chiamate la funzione, il C++ genera automaticamente le altre funzioni generiche, in modo da trattare correttamente ogni tipo di chiamata. Quindi in definitiva, definendo una solo funzione generica, avrete definito un'intera famiglia di soluzioni.

Tutte le definizioni di funzioni generiche iniziano con la parola riservata `template` seguita da una lista di parametri formali di tipo racchiusi tra parentesi angolari (< e >). Ciascun parametro formale di tipo è preceduto dalla parola riservata `class`. I parametri formali di tipo sono tipi predefiniti o definiti dall'utente che specificano il tipo degli argomenti di una funzione e il tipo restituito dalla funzione e servono a dichiarare le variabili all'interno del corpo della funzione. A parte ciò, la definizione della funzione procede allo stesso modo di quella di qualsiasi altra funzione.

In Figura 3.27 è utilizzato il seguente template di funzione:

```
template <class T>
T maximum( T value1, T value2, T value3 )
{
   T max = value1;

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   return max;
}
```

Questo template dichiara un solo parametro di tipo formale T, indicando al compilatore il tipo di dato da verificare nella funzione `maximum`. Quando si invoca la funzione `maximum` il tipo di dato passato a `maximum` sostituisce T in tutto il template, e al termine il compilatore avrà creato una funzione completa per determinare il massimo fra tre valori del tipo di dato specificato. La nuova funzione sarà poi compilata: i template sono quindi un mezzo per generare nuovo codice.

Nella Figura 3.27 sono istanziate tre funzioni: una per i valori di tipo `int`, un'altra per i `double` e l'ultima per i `char`. L'istanza della funzione per i valori di tipo `int` è come segue:

```
int maximum( int value1, int value2, int value3 )
{
   int max = value1;

   if ( value2 > max )
      max = value2;
   if ( value3 > max )
      max = value3;

   return max;
}
```

Ciascun parametro di tipo presente nella definizione del template deve comparire almeno una volta nella lista dei parametri. Il nome di un parametro di tipo deve essere unico all'interno della lista di parametri formali di un particolare template.

La Figura 3.27 mostra l'utilizzo della funzione generica `maximum` per determinare il maggiore fra tre valori di tipo `int`, poi di tipo `double` e infine di tipo `char`.

```
1   // Fig. 3.27: fig03_27.cpp
2   // Utilizzo di un template di funzione
3   #include <iostream.h>
4
5   template < class T >
6   T maximum( T value1, T value2, T value3 )
7   {
8      T max = value1;
9
10     if ( value2 > max )
11        max = value2;
12
13     if ( value3 > max )
14        max = value3;
15
16     return max;
17  }
18
19  int main()
20  {
21     int int1, int2, int3;
22
23     cout << "Input three integer values: ";
24     cin >> int1 >> int2 >> int3;
25     cout << "The maximum integer value is: "
26          << maximum( int1, int2, int3 );        // versione int
27
28     double double1, double2, double3;
```

Figura 3.27 Utilizzo di un template di funzione (continua)

```
29
30      cout << "\nInput three double values: ";
31      cin >> double1 >> double2 >> double3;
32      cout << "The maximum double value is: "
33         << maximum( double1, double2, double3 ); // versione double
34
35      char char1, char2, char3;
36
37      cout << "\nInput three characters: ";
38      cin >> char1 >> char2 >> char3;
39      cout << "The maximum character value is: "
40           << maximum( char1, char2, char3 )       // versione char
41           << endl;
42
43      return 0;
44   }
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3
Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3
Input three characters: A C B
The maximum character value is: C
```

Figura 3.27 Utilizzo di un template di funzione.

Errore tipico 3.35

Se dimenticate la parola riservata `class` *prima di ogni parametro di tipo nel template di una funzione commettete un errore di sintassi.*

Errore tipico 3.36

Se non utilizzate tutti i parametri di tipo del template nella segnatura di una funzione commettete un errore di sintassi.

3.22 Pensare in termini di oggetti: come identificare gli attributi di una classe [progetto opzionale]

Nella sezione "Pensare in termini di oggetti" del Capitolo 2 abbiamo introdotto la prima fase della progettazione orientata agli oggetti (OOD) di un simulatore di ascensore, che è consistita nell'identificare le classi da implementare. Abbiamo cominciato elencando i nomi presenti nella definizione del problema e creando una classe distinta per ogni categoria di nomi relativa a un compito rilevante del simulatore. Abbiamo poi rappresentato le classi e le loro relazioni in un diagramma delle classi UML. Le classi sono composte da *attributi* e *operazioni*; nei programmi C++ gli attributi sono implementati come dati mentre le operazioni come funzioni. In questo capitolo determineremo la maggior parte degli attributi delle classi del nostro simulatore, nel Capitolo 4 passeremo a considerare le operazioni e nel Capitolo 5 ci concentreremo sulle interazioni tra i vari oggetti del simulatore, dette anche *collaborazioni*. Considerate gli attributi di alcuni oggetti del mondo reale. Tra gli attributi di una persona possiamo considerare l'altezza e il peso. Fra quelli di una radio possiamo pensare alla stazione sintonizzata, al volume, e all'impostazione della banda (AM o FM). Gli attributi di un'automobile includono il valore corrente di velocità e il numero di chilometri percorsi, il valore del livello del carburante e la marcia inserita. Per un personal computer potremmo considerare il produttore (Apple, IBM o Compaq, ad esempio), il tipo di schermo (monocromatico o a colori), la quantità di memoria centrale (in Megabyte) e la capacità dell'hard disk (in Gigabyte).

Gli attributi descrivono le classi, per cui possiamo individuarli cercando le parole e le locuzioni descrittive nella definizione del problema. Per ognuna di esse possiamo quindi creare un attributo e assegnarlo alla classe. Creeremo anche gli attributi che rappresentano i dati necessari alla classe: per esempio, la classe `Scheduler` ha bisogno di conoscere gli istanti temporali in cui creare la persona successiva su uno dei piani. La tabella in Figura 3.28 elenca le parole e le locuzioni descrittive di ciascuna classe.

Come vedete, le classi `Bell` e `Building` non hanno attributi. Andando avanti con il nostro simulatore aggiungeremo, modificheremo e cancelleremo di continuo delle informazioni su ogni classe del sistema.

Classe	Parole e locuzioni descrittive
`Elevator`	all'inizio della giornata è in attesa (*primo piano*) alterna le corse (su e giù) capacità di 1 persona impiega 5 secondi per andare da un piano all'altro l'ascensore si sposta
`Clock`	all'inizio della giornata il timer è impostato a 0
`Scheduler`	[programma l'arrivo della prossima persona al piano] in un arco da 5 a 20 secondi nel futuro a partire dal momento attuale (per ciascun piano)
`Floor`	capacità di 1 persona occupato / libero
`FloorButton`	premuto
`ElevatorButton`	*premuto*
`Door`	chiusa / aperta
`Bell`	*nessuna*
`Light`	accesa / spenta
`Building`	*nessuna*

Figura 3.28 Parole e locuzioni descrittive nella definizione del problema.

La Figura 3.29 contiene il diagramma delle cassi che elenca alcuni attributi di ogni classe, creati dalle parole e locuzioni descrittive della Figura 3.28. Nel diagramma delle classi UML, gli attributi di una classe si scrivono nella porzione centrale del rettangolo di una classe. Consideriamo il seguente attributo della classe `Elevator`:

```
capacity : int = 1
```

Da questo elenco possiamo osservare che ciascun'attributo ha tre caratteristiche. Innanzitutto esso ha un *nome* (`capacity`); poi ha un *tipo* (`int`), che dipende dal linguaggio di programmazione in cui si implementa il sistema(per esempio in C++ il valore può essere un tipo primitivo, come `int`, `char` o `float`, o un tipo definito dall'utente, come una classe: vedremo nel Capitolo 6 che ogni classe è sostanzialmente un nuovo tipo di dato).Infine, un attributo può avere un *valore iniziale*. Nel caso di `capacity`, esso è definito ed è pari a 1. Se un attributo non ha un valore iniziale, esso viene semplicemente omesso. Per esempio l'attributo `floor1ArrivalTime` (istante di arrivo al primo piano) della classe `Scheduler` sarà di tipo `int`, ma non possiamo associargli alcun valore iniziale, perché il valore di questo attributo è un numero casuale che sarà generato soltanto durante l'esecuzione. Per ora non è il caso di preoccuparci eccessivamente del tipo e del valore iniziale degli attributi, includiamo soltanto le nozioni che possiamo estrarre direttamente dalla definizione del problema.

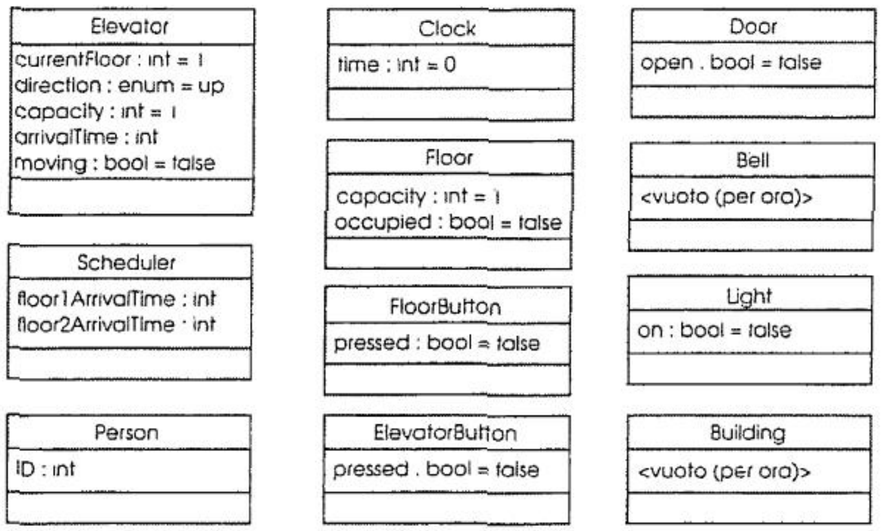


Figura 3.29 Diagramma delle classi con gli attributi.

3.22.1 I diagrammi di stato

Gli oggetti di un sistema in un dato istante si trovano in uno *stato* che può modificarsi nel corso del tempo. I *diagrammi di stato* sono un modo per esprimere come e in quali condizioni gli oggetti di un sistema cambiano il loro stato.

La Figura 3.30 è un semplice diagramma di stato per gli oggetti della classe `FloorButton` o `ElevatorButton`. Ogni stato è rappresentato da un rettangolo arrotondato che ne contiene il nome, e un cerchietto pieno con una freccia punta allo stato iniziale ("non premuto" in questo caso). La linea intera con le frecce indica la *transizione* tra gli stati. Un oggetto può effettuare una transizione da uno stato all'altro in risposta a un *evento*: per esempio le classi `FloorButton` ed `ElevatorButton` cambiano stato, da "non premuto" a "premuto".

in risposta all'evento "premere pulsante". Il nome dell'evento che causa la transizione viene riportato vicino alla linea corrispondente e, come vedremo, potrà includere anche altre informazioni sugli eventi.

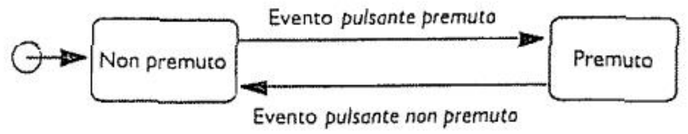


Figura 3.30 Diagramma di stato delle classi FloorButton ed ElevatorButton.

La Figura 3.31 mostra il diagramma di stato della classe `Elevator`. In un qualunque istante, l'ascensore può trovarsi solo in uno dei seguenti stati: "in attesa", "in servizio al piano" (ovvero fermo al piano ma occupato ad azzerare il pulsante interno o a comunicare con il piano) e, infine, "in corsa". All'inizio della simulazione l'ascensore si trova nello stato "in attesa". Gli eventi che causano le transizioni sono indicati in corrispondenza delle relative linee di transizione.

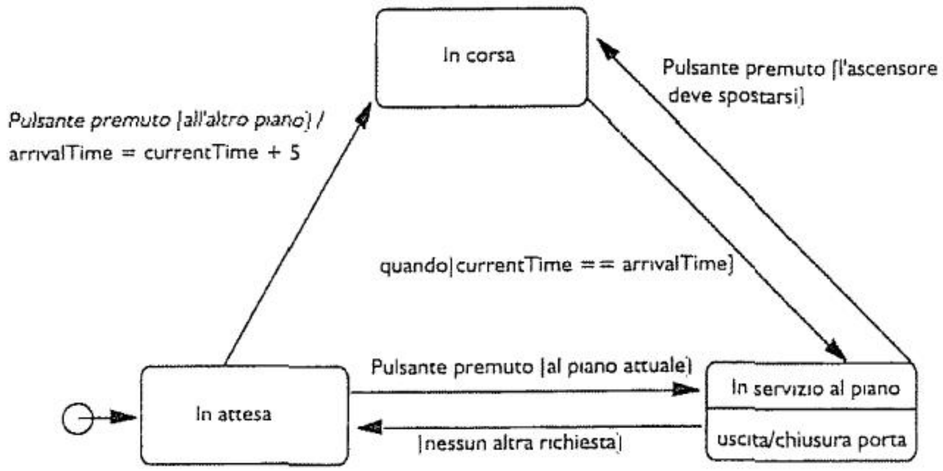


Figura 3.31 Diagramma di stato della classe Elevator.

Analizziamo ora gli eventi presenti in questo diagramma degli stati. Il testo

```
Premere il pulsante [occorre spostamento]
```

Ci dice che l'evento "pulsante premuto" fa sì che l'ascensore effettui la transizione da "in servizio al piano" a "in corsa". La *condizione di guardia* (posta fra parentesi quadrate) indica che la transizione si può verificare unicamente se l'ascensore ha bisogno di spostarsi. Il testo dell'evento completo dice che l'ascensore passa dallo stato "in servizio al piano" allo stato "in corsa" in risposta all'evento "pulsante premuto" soltanto se ha bisogno di effettuare uno spostamento. Allo stesso modo, l'ascensore passa da "in attesa" a "in servizio al piano" quando viene premuto il pulsante esterno del piano in cui l'ascensore sosta.

Il testo corrispondente alla linea di transizione dallo stato "in attesa" allo stato "in corsa" indica che questa transizione si verifica con l'evento "pulsante premuto" [all'altro piano]. La barra ("/") indica che questo cambiamento di stato è accompagnato da un'*azio-*

c) La funzione `instructions` che non riceve alcun argomento e non restituisce alcun valore. Nota: funzioni di questo genere sono usate frequentemente per visualizzare agli utenti delle istruzioni.
d) La funzione `intToFloat` che prende l'argomento intero `number`, e restituisce un risultato in virgola mobile.

3.5 Scrivete il prototipo delle seguenti funzioni:

a) La funzione dell'Esercizio 3.4a.
b) La funzione dell'Esercizio 3.4b.
c) La funzione dell'Esercizio 3.4c.
d) La funzione dell'Esercizio 3.4d.

3.6 Scrivete le dichiarazioni relative a:

a) L'intero `count` che dovrebbe essere conservato in un registro. Inizializzate `count` a 0.
b) La variabile a virgola mobile `lastVal` che deve conservare il suo valore anche per le chiamate successive alla funzione in cui è definita.
c) L'intero esterno `number` il cui scope dovrebbe essere limitato al resto del file in cui è definito.

3.7 Trovate l'errore in ognuno di questo segmenti di programma e indicate come correggerlo (cfr. anche Esercizio 3.53):

a)
```
int g( void ) ;
   cout << "Inside function g" << endl;
   int h( void ) {
      cout << "Inside function h" << endl;
   }
```

b)
```
int sum( int x, int y ) ;
   int result;

   result = x + y;
}
```

c)
```
int sum( int n ) {
   if ( n == 0 )
      return 0;
   else
      n + sum( n     );
}
```

d)
```
void f( float a ); ;
   float a;

   cout << a << endl;
}
```

e)
```
void product( void ) {
  int a, b, c, result;
   cout << "Enter three integers: ",
   cin >> a >> b >> c;
   result = a * b * c;
   cout << "Result is " << result;
   return result;
}
```

3.8 Cosa significa l'indicazione `float&` in un prototipo di funzione?

3.9 (Vero/Falso) Tutte le chiamate in C++ sono effettuate per valore.

3.10 Scrivete un programma completo in C++ che utilizza la funzione `inline` di nome `sphereVolume`, la quale chiede all'utente il raggio di una sfera e ne calcola il volume secondo la formula:

```
volume = ( 4 / 3 ) * 3.14159 * pow( radius, 3 ).
```

Risposte agli esercizi di autovalutazione

3.1 a) Funzioni e classi. b) Chiamata di funzione. c) Variabile locale. d) `return`. e) `void` f) Scope. g) `return;` o `return espressione;` o la parentesi graffa chiusa che termina la funzione. h) Prototipo di funzione. i) `rand` j) `srand`. k) `auto`, `register`, `extern`, `static`. l) `auto`. m) `register`. n) esterna, globale. o) `static`. p) Scope a livello funzione, di file, di blocco, di prototipo di funzione. q) Ricorsiva. r) Base. s) Overloading. t) Operatore unario di risoluzione dello scope (`::`). u) `const`. v) Template.

3.2 a) Scope a livello di blocco. b) Scope a livello di blocco. c) Scope a livello di file. d) Scope a livello di file. e) Scope a livello di file. f) Scope a livello di prototipo di funzione.

3.3 Come segue.

```
1  // ex03_03.cpp
2  /* Verifica delle funzioni della libreria math */
3  #include <iostream.h>
4  #include <iomanip.h>
5  #include <math.h>
6
7  int main()
8  {
9     cout << setiosflags( ios::fixed | ios::showpoint )
10          << setprecision( 1 )
11          << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
12          << "\nsqrt(" << 9.0 << ") = " << sqrt( 9.0 )
13          << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
14          << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
15          << ") = " << setprecision( 6 ) << exp( 2.0 )
16          << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
17          << log( 2.718282 ) << "\nlog(" << setprecision( 6 )
18          << 7.389056 << ") = " << setprecision( 1 )
19          << log( 7.389056 ) << endl;
20     cout << "log10(" << 1.0 << ") = " << log10( 1.0 )
21          << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
22          << "\nlog10(" << 100.0 << ") = " << log10( 100.0 )
23          << "\nfabs(" << 13.5 << ") = " << fabs( 13.5 )
24          << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
25          << "\nfabs(" << -13.5 << ") = " << fabs( -13.5 ) << endl;
26     cout << "ceil(" << 9.2 << ") = " << ceil( 9.2 )
27          << "\nceil(" << -9.8 << ") = " << ceil( -9.8 )
28          << "\nfloor(" << 9.2 << ") = " << floor( 9.2 )
29          << "\nfloor(" << -9.8 << ") = " << floor( -9.8 ) << endl;
30     cout << "pow(" << 2.0 << ", " << 7.0 << ") = "
31          << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
32          << 0.5 << ") = " << pow( 9.0, 0.5 )
33          << setprecision(3) << "\nfmod("
34          << 13.675 << ", " << 2.333 << ") = "
35          << fmod( 13.675, 2.333 ) << setprecision( 1 )
36          << "\nsin(" << 0.0 << ") = " << sin( 0.0 )
37          << "\ncos(" << 0.0 << ") = " << cos( 0.0 )
38          << "\ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
39     return 0;
40  }
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

3.4 a) `double hypotenuse( double side1, double side2 )`
b) `int smallest( int x, int y, int z )`
c) `void instructions( void )  // in C++ (void) si puo scrivere come ()`
d) `float intToFloat( int number )`

3.5 a) `double hypotenuse( double, double );`
b) `int smallest( int, int, int );`
c) `void instructions( void ); // in C++ (void) si può scrivere come ()`
d) `float intToFloat( int );`

3.6 a) `register int count = 0;`
b) `static float lastVal;`
c) `static int number;`
Nota: Deve comparire al di fuori di ogni definizione di funzione.

3.7 a) Errore: La funzione `h` è definita nella funzione `g`.
Correzione: Spostare `h` al di fuori della definizione di `g`.

b) Errore: La funzione deve ritornare un intero, ma non è così.
Correzione: Eliminare la variabile `result` e scrivere la seguente istruzione:
`return x + y;`

c) Errore: La funzione non restituisce il risultato di `n + sum(n - 1);` `sum` restituisce un risultato scorretto.
Correzione: Riscrivere l'istruzione della clausola `else` come segue:
`return n + sum(n - 1);`

d) Errore: C'è un punto e virgola dopo la parentesi destra della lista dei parametri, e il parametro `a` viene ridefinito nella definizione della funzione.
Correzione: Eliminare il punto e virgola e la dichiarazione `float a;`.

e) Errore: La funzione restituisce un valore, ma non dovrebbe.
Correzione: Eliminare l'istruzione `return`.

3.8 Significa che il programmatore dichiara un parametro come "riferimento a un tipo `float`", per accedere alla variabile originaria che viene passata come parametro.

3.9 Falso. Il C++ consente la chiamata diretta per riferimento, tramite i riferimenti e l'utilizzo dei puntatori.

3.10 Come segue.

```
1  // ex03_10.cpp
2  // Inline function that calculates the volume of a sphere
3  #include <iostream.h>
4
5  const float PI = 3.14159;
6
7  inline float sphereVolume( const float r )
8     { return 4.0 / 3.0 * PI * r * r * r; }
9
10 int main()
11 {
12    float radius;
13
14    cout << "Enter the length of the radius of your sphere: ";
15    cin >> radius;
16    cout << "Volume of sphere with radius " << radius <<
17            " is " << sphereVolume( radius ) << endl;
18    return 0;
19 }
```

Esercizi

3.11 Qual è il valore di `x` dopo ciascun assegnamento?

a) `x = fabs( 7.5 )`
b) `x = floor( 7.5 )`
c) `x = fabs( 0.0 )`
d) `x = ceil( 0.0 )`
e) `x = fabs( -6.4 )`
f) `x = ceil( -6.4 )`
g) `x = ceil( -fabs( -8 + floor( 5.5 ) ) )`

3.12 Un autosilo chiede un minimo di $2.00 per parcheggiare fino a tre ore; per ogni ora successiva, l'autosilo chiede $0.50 all'ora. La somma richiesta per una sosta di 24 ore ammonta a $10.00. Ipotizziamo che nessuna macchina rimanga parcheggiata per più di 24 ore. Scrivete un programma che calcola quanto devono pagare tre clienti dell'autosilo per la giornata appena trascorsa. Per ogni cliente dovreste immettere il numero di ore di parcheggio. Il programma visualizzerà i risultati in formato tabulare, calcolando anche il totale dei guadagni relativi alla giornata trascorsa. La somma dovuta da ogni cliente viene calcolata dalla funzione `calculateCharges`. L'output dovrebbe essere del tipo:

```
Macchina    Ore     A pagare
1           1.5         2.00
2           4.0         2.50
3          24.0        10.00
TOTAL      29.5        14.50
```

3.13 Un'applicazione della funzione `floor` è l'arrotondamento di un valore all'intero più vicino. L'istruzione

```
y = floor( x + 0.5 );
```

arrotonda `x` all'intero più vicino e assegna il risultato a `y`. Scrivete un programma che legge una sequenza di numeri e utilizza l'istruzione precedente per arrotondarli all'intero più vicino. Per ogni numero che elaborate, visualizzate il numero originario e l'arrotondamento.

3.14 La funzione `floor` può essere utilizzata per arrotondare un numero a una posizione decimale specifica. L'istruzione

```
y = floor( x * 10 + .5 ) / 10;
```

arrotonda `x` alla prima posizione decimale (i decimi). L'istruzione

```
y = floor( x * 100 + .5 ) / 100;
```

lo arrotonda alla seconda posizione decimale (i centesimi). Scrivete un programma che definisce quattro funzioni per arrotondare un numero in vari modi:

a) `roundToInteger( number )` `// arrotondamento a un intero`
b) `roundToTenths( number )` `// alla prima cifra decimale`
c) `roundToHundredths( number )` `// alla seconda`
d) `roundToThousandths( number )` `// alla terza`

Per ogni valore letto in input il programma fornirà le quattro versioni di arrotondamento.

3.15 Rispondete alle seguenti domande.

a) Che significa scegliere un numero "a caso"?
b) Perché la funzione `rand` può servire a simulare i giochi d'azzardo?
c) Che bisogno c'è di utilizzare la funzione `srand`? In quali circostanze non è il caso di utilizzarla?
d) Perché spesso bisogna scalare o traslare i valori generati da `rand`?
e) A che serve simulare al computer gli eventi del mondo reale?

3.16 Scrivete delle istruzioni che assegnano alla variabile n dei valori casuali nei seguenti intervalli:

a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$
e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

3.17 Dati questi insiemi di interi, scrivete un'istruzione che estragga casualmente i numeri compresi in ciascuno di essi.

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

3.18 Scrivete la funzione `integerPower(base, exponent)` che restituisce il valore di

$base^{exponent}$

Per esempio `integerPower(3,4) = 3 * 3 * 3 * 3`. Supponete che l'esponente sia un intero positivo diverso da zero, e che la base sia un intero. La funzione `integerPower` dovrebbe controllare il calcolo con un ciclo `for` o `while`. Non utilizzate alcuna funzione matematica di libreria.

3.19 Definite la funzione `hypotenuse` che calcola la lunghezza dell'ipotenusa di un triangolo rettangolo, a partire dalla lunghezza dei due cateti. Utilizzate questa funzione per determinare l'ipotenusa dei seguenti triangoli; la funzione prende due valori di tipo `double` e restituisce un valore `double`.

Triangolo	Cateto 1	Cateto 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

3.20 Scrivete la funzione `multiple` che riceve una coppia di interi e determina se il secondo è multiplo del primo. La funzione riceve due interi e restituisce `true` se il secondo è multiplo del primo, altrimenti restituisce `false`. Utilizzate questa funzione in un programma che legga in input una serie di coppie di interi.

3.21 Scrivete un programma che legge in input una serie di interi e li passa uno alla volta alla funzione `even`, che per ognuno di essi determina se è un numero pari (con l'operatore `%`). La funzione riceve un argomento intero e restituisce `true` se esso è pari, `false` altrimenti.

3.22 Scrivete una funzione che visualizza un quadrato di asterischi allineato al margine sinistro dello schermo. Il lato del quadrato è un intero passato come argomento. Se il lato vale 4, per esempio, la funzione visualizza

```
****
****
****
****
```

3.23 Modificate la funzione dell'Esercizio 3.22 per creare un quadrato formato da un carattere passato nel parametro `fillCharacter`. Se il lato del quadrato è 5 e `fillCharacter` è `'#'` la funzione deve visualizzare

```
#####
#####
#####
#####
#####
```

3.24 Utilizzate le tecniche che avete applicato negli Esercizi 3.22 e 3.23 per scrivere programmi che generano un'ampia gamma di figure.

3.25 Scrivete dei segmenti di programma che fanno ciò che segue:

a) Calcolano la parte intera del quoziente, quando l'intero `a` viene diviso per l'intero `b`.
b) Calcolano il resto intero quando l'intero `a` viene diviso per l'intero `b`.
c) Utilizzano i segmenti che avete scritto per le parti a) e b) per scrivere una funzione che legge in input un intero tra 1 e 32767 e lo visualizza come una serie di cifre, separandone ogni coppia con due spazi. Per esempio, l'intero 4562 sarebbe visualizzato come

```
4 5 6 2
```

3.26 Scrivete una funzione che riceve come argomento un orario decomposto in tre numeri interi (ore, minuti e secondi) e restituisce il numero di secondi che sono passati dall'ultima volta che l'orologio ha battuto le ore 12. Utilizzate questa funzione per determinare quanto tempo passa tra due orari dati, su un orologio a lancette col quadrante di 12 ore.

3.27 La scala Fahrenheit è una scala di temperatura utilizzata negli Stati Uniti che divide l'intervallo fra il punto di fusione e quello di ebollizione dell'acqua in 180 gradi e assegna il valore di 32°F al punto di fusione dell'acqua. La scala comunemente utilizzata in Europa, invece, è la scala Celsius che assegna il valore 0°C alla punto di fusione dell'acqua ed il valore 100°C al valore di ebollizione.

Dunque, la conversione fra gradi Celsius e Farenheit può essere effettuata osservando che 1°C ≈ 180/100 = 9/5°F e che 0°C = 32°F. L'esercizio vi richiede di implementare le seguenti funzioni a valori interi:

a) La funzione `celsius` restituisce l'equivalente Celsius di una temperatura espressa in gradi *Fahrenheit*.
b) La funzione `fahrenheit` restituisce l'equivalente Fahrenheit di una temperature espressa in gradi Celsius.
c) Utilizzate queste due funzioni per scrivere un programma che visualizza un grafico che mostra i valori Fahrenheit equivalenti ai gradi Celsius da 0 a 100. Visualizzate il risultato in formato tabulare, minimizzando il numero di linee di output, ma senza sacrificare la leggibilità.

3.28 Scrivete una funzione che restituisce il più piccolo di tre numeri a virgola mobile.

3.29 Un numero intero è detto *perfetto* se esso è uguale alla somma dei suoi fattori, incluso 1 ed (ovviamente) escluso il numero stesso. Per esempio 6 è un numero perfetto, poiché 6 = 1 + 2 + 3. Scrivete la funzione `perfect` che determina se il numero passato come parametro è perfetto. Utilizzate questa funzione in un programma che determina tutti i numeri perfetti compresi tra 1 e 1000. Per ogni numero perfetto trovato, visualizzatene anche i fattori, in modo che possiate rendervi conto se la funzione è stata scritta correttamente. Se volete, potete trovare anche i numeri perfetti che superano la soglia del primo migliaio (attenzione, in questo caso ci potrebbe volere molto tempo prima che il programma termini).

3.30 Un numero intero si dice *primo* se è divisibile soltanto per 1 e per se stesso. Per esempio, 2, 3, 5 e 7 sono numeri primi, mentre non lo sono 4, 6, 8 e 9.

a) Scrivete una funzione che determina se il suo parametro è un numero primo.
b) Utilizzate questa funzione per scrivere un programma che determina e visualizza i numeri primi compresi tra 1 e 10000. Quanti numeri avete bisogno di provare realmente prima di trovare tutti i numeri primi di questo intervallo?
c) Inizialmente penserete che non serve andare oltre *n/2* per verificare se *n* è un numero primo, ma in realtà potete anche fermarvi alla sua radice quadrata. Perché? Riscrivete il programma ed eseguite le due versioni. Che miglioramento avete ottenuto sulle prestazioni?

3.31 Scrivete una funzione che riceve un argomento intero e restituisce il numero che ha le cifre al contrario. Per esempio, dato il numero 7631, la funzione restituirà 1367.

3.32 Il *massimo comun divisore* (MCD) di due interi è il più grande valore intero che divide i due numeri. Ad esempio, il MCD di 9 e 12 è 3 e quello di 5 e 9 è 1. Scrivete la funzione `mcd` che restituisce il MCD di due numeri interi.

3.33 Scrivete la funzione `qualityPoints` che riceve in input la media di uno studente e restituisce 4 se la media è nell'intervallo 90-100, 3 se è nell'intervallo 80-89, 2 se è nell'intervallo 70-79, 1 se è nell'intervallo 60-69 e 0 se è più bassa di 60.

3.34 Scrivete un programma che simula il lancio di una moneta. Per ogni lancio il programma dovrebbe visualizzare `testa` o `croce`. Per ogni esecuzione effettuate 100 lanci e contate il numero di teste e croci che sono uscite e visualizzatene il risultato. Il programma dovrebbe chiamare la funzione `flip` che non prende argomenti e restituisce 0 per croce e 1 per testa. Nota: se la simulazione è realistica (cioè se il vostro generatore di numeri casuali funziona bene), ogni lato della monete dovrebbe apparire all'incirca per metà dei lanci.

3.35 I computer si stanno facendo strada prepotentemente negli ambiti educativi. Scrivete un programma che aiuta un alunno della scuola elementare a imparare la moltiplicazione. Utilizzate `rand` per generare due interi positivi di una cifra sola. Il programma dovrebbe porre all'alunno una domanda del tipo:

```
Quanto fa 6 per 7?
```

L'alunno digita la risposta. Il programma la controlla e, se è corretta, visualizza `"Ottimo! La risposta è corretta!"` e passa alla domanda successiva. Se la risposta è sbagliata il programma visualizzerà `"Mi dispiace, la risposta è sbagliata. Riprova!"` e consente all'alunno di riprovare, finché (si spera) non dà la risposta corretta.

3.36 L'utilizzo dei computer per scopi educativi è noto anche come *istruzione assistita dal calcolatore* (in inglese, CAI: *Computer Aided Instruction*). Uno dei problemi più frequenti in questo ambito è la distrazione e la stanchezza dello studente, ma può essere evitato variando l'interazione con lo studente, in modo che l'attenzione sia sempre desta. Modificate il programma dell'Esercizio 3.35 in modo che per ogni risposta corretta o sbagliata il programma vari il commento, un po' come segue:

Risposte corrette

```
Bravo!
Eccellente!
Ottimo lavoro!
Continua così!
```

Risposte sbagliate

```
No. Riprova!
Sbagliato. Prova ancora!
Non ti arrendere, riprova!
Niente da fare. Prova ancora!
```

Utilizzate i numeri casuali per scegliere la frase, e un costrutto `switch` per visualizzarla.

3.37 Sistemi di istruzione assistita più complessi elaborano delle statistiche sulle prestazioni degli studenti nell'arco di un periodo di tempo. La decisione di avanzare all'argomento successivo si basa *sulla comprensione dello studente di tutti quelli precedenti*. Modificate il programma dell'Esercizio 3.36 per contare il numero di risposte giuste e sbagliate. Quando lo studente ha digitato 10 risposte, il programma deve calcolare la percentuale di risposte corrette. Se le risposte giuste sono meno del 75%, il programma dovrebbe visualizzare `"Chiedete un aiuto al vostro istruttore!"` e terminare l'esecuzione.

3.38 Scrivete un programma che gioca a `"Indovina il numero!"`. Il programma sceglie un numero a caso tra 1 e 1000, poi visualizza:

```
Ho pensato un numero tra 1 e 1000.
Prova un po' a indovinarlo!
Digita il tuo primo tentativo:
```

Il giocatore digita il tentativo. Il programma quindi lo valuta e visualizza una frase tra le seguenti:

```
1. Bravissimo! Hai indovinato il numero!
   Vuoi giocare ancora (s/n)?
2. Troppo basso. Riprova.
3. Troppo alto. Riprova.
```

Se il numero digitato dal giocatore è sbagliato, il programma dovrebbe continuare a concedere un nuovo tentativo, finché non viene indovinato. Il programma deve poi continuare a indicare se il numero è troppo alto o troppo basso, finché il giocatore non becca quello giusto. Nota: la tecnica di ricerca impiegata nel programma è detta *ricerca binaria*: ne parleremo più diffusamente nel prossimo problema.

3.39 Modificate il programma dell'Esercizio 3.38 per contare il numero di tentativi effettuati dal giocatore. Se il numero è minore di 10, visualizzate `"O conosci il segreto o sei nato con la camicia!"`, se è uguale a 10 `"Ah! Conosci il segreto!"` e se è maggiore `"Puoi fare di meglio"` Perche abbiamo centrato tutto su 10 tentativi? Perché per ogni buon tentativo l'utente dovrebbe essere in grado di eliminare metà dei numeri. Quindi per indovinare un numero da 1 a 1000 non dovrebbero servire più di 10 tentativi.

3.40 Scrivete la funzione ricorsiva `power (base, exponent)` che restituisce

$base^{exponent}$

Per esempio, `power(3, 4) = 3 * 3 * 3 * 3`. Si suppone che *exponent* sia un intero maggiore o uguale a 1. Suggerimento: Il passo di ricorsione utilizzerà la relazione

$Base^{exponent} = base \cdot base^{exponent - 1}$

e la condizione per terminare la ricorsione si verifica quando *exponent* vale 1 perché

$base^1 = base$

3.41 La serie di Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inizia con i termini 0 e 1 e ha la proprietà che ogni termine successivo è la somma dei due che lo precedono. a) Scrivete la funzione non ricorsiva `fibonacci(n)` che calcola l'*n*-esimo numero di Fibonacci. b) Determinate il più grande numero di Fibonacci che può essere visualizzato sul vostro sistema. Modificate il programma della parte a) per utilizzare un tipo `double` anziche un tipo `int`.

3.42 (*Torre di Hanoi*) Qualsiasi informatico *in statu nascendi* deve cimentarsi con una serie di problemi classici e tra questi è impossibile non imbattersi nel celeberrimo problema della Torre di Hanoi (Figura 3.33). Si tramanda che in un antico tempio orientale i monaci tentassero di spostare una serie di dischi da un paletto a un altro. Nel paletto iniziale sono infilati 64 dischi, disposti in ordine decrescente di dimensione. I monaci debbono trasferire la pila di dischi su un altro paletto spostando un disco per volta e non potendo mai mettere un disco più largo su uno più piccolo. I monaci hanno a disposizione un terzo paletto sul quale spostare temporaneamente i dischi. Dato che si crede che il mondo finirà quando i monaci avranno terminato questo compito, non abbiamo un gran incentivo a dare il nostro contributo ai loro sforzi.

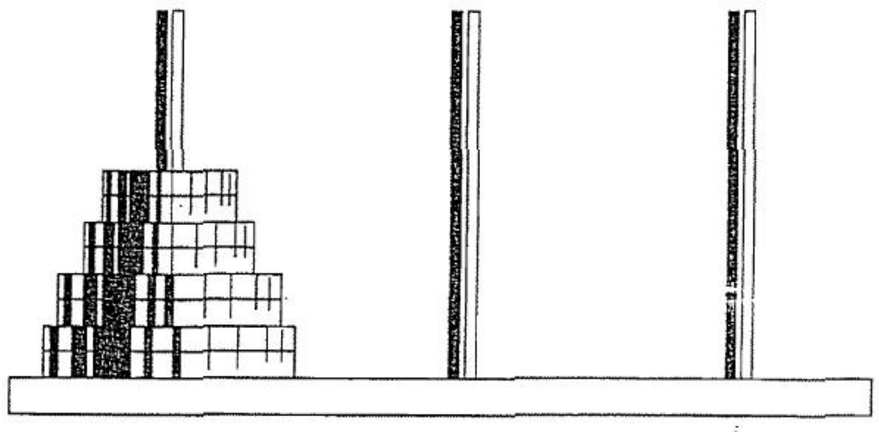


Figura 3.33 Le torri di Hanoi nel caso dei quattro dischi.

Adesso supponiamo che i monaci vogliano spostare i dischi dal paletto 1 al paletto 3. Vogliamo sviluppare un algoritmo che visualizzi la sequenza precisa degli spostamenti da paletto a paletto.
Se dovessimo scegliere un approccio convenzionale per questo problema, ci troveremmo subito in difficoltà, con i dischi sparsi in disordine sui paletti. Invece se affrontiamo il problema utilizzando la ricorsione, esso diventa immediatamente più facilmente trattabile: lo spostamento di *n* dischi si può vedere in termini dello spostamento di soli $n - 1$ dischi, da cui la ricorsione. In che modo? È presto detto:

a) Spostare $n - 1$ dischi dal paletto 1 al paletto 2, utilizzando il paletto 3 come area temporanea.
b) Spostare l'ultimo disco (il più grande) dal paletto 1 al paletto 3.
c) Spostare $n - 1$ dischi dal paletto 2 al paletto 3, utilizzando il paletto 1 come area temporanea.

Il procedimento termina quando il compito si riduce al caso base, cioè quando *n* vale 1. A questo punto basta spostare il disco verso la sua destinazione, senza bisogno di utilizzare l'area temporanea.
Scrivete un programma che risolve il problema della Torre di Hanoi. Utilizzate una funzione ricorsiva con quattro parametri:

a) Il numero di dischi da spostare
b) Il paletto in cui sono infilati inizialmente i dischi
c) Il paletto su cui deve essere spostata la pila di dischi
d) Il paletto da utilizzare come area temporanea

Il programma deve visualizzare le precise istruzioni per spostare i dischi dal paletto iniziale al paletto finale. Per esempio, per spostare tre dischi dal paletto 1 al paletto 3, ecco quali sono le mosse necessarie:

1 → 3 (Indica lo spostamento di un disco dal paletto 1 al paletto 3)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3 (anche qui: controlla che tra i numeri rimanga la freccia)

3.43 Qualsiasi programma che può essere implementato ricorsivamente, può anche essere implementato iterativamente, anche se a volte ciò comporta una maggiore difficoltà e una minore chiarezza del codice. Provate a scrivere una versione iterativa della Torre di Hanoi. Se ci riuscite, confrontate le due versioni del programma.

3.44 (*Visualizzazione della ricorsione*) È interessante osservare la ricorsione "in azione". Modificate la funzione `factorial` in Figura 3.14 in modo da visualizzare la variabile locale e il parametro della chiamata ricorsiva. Per ogni chiamata ricorsiva, visualizzate l'output su una linea diversa, con un ulteriore livello di rientro. Fate del vostro meglio per rendere l'output interessante e significativo. Il vostro obiettivo qui è progettare e implementare un formato di visualizzare la ricorsione, per comprendere meglio come avviene.

3.45 Il massimo comun divisore degli interi `x` e `y` è il ... che divide i due numeri. Scrivete la funzione ricorsiva `mcd` che restituisce il massimo comun divisore di `x` ... può essere definito ricorsivamente come segue: se `y` è uguale a 0, allora `mcd(x, y)` è `x`; altrimenti `mcd(x, y)` è `mcd(y, x % y)` dove `%` è l'operatore modulo.

3.46 Si può chiamare `main` ricorsivamente? Scrivete un programma che lo fa. Nella funzione includete la variabile locale statica `count` inizializzata a 1. Postincrementate e visualizzate il valore di `count` ogni volta che viene chiamata `main`. Che cosa succede durante l'esecuzione?

3.47 Negli Esercizi dal 3.35 al 3.37 avete scritto un programma di apprendimento assistito dal calcolatore per insegnare la moltiplicazione a un alunno delle scuole elementari. In questo esercizio cercheremo di migliorare quel programma.

a) Modificate il programma prevedendo diversi livelli di abilità. Per esempio, al livello 1 l'alunno dovrà moltiplicare numeri di 1 sola cifra, al livello 2 di 2 cifre, e così via.
b) Modificate il programma per consentire all'utente di scegliere il tipo di operazione matematica su cui cimentarsi. L'opzione 1 significherà addizione, 2 sottrazione, 3 moltiplicazione, 4 divisione e 5 un'operazione a caso, scelta fra le prime quattro.

3.48 Scrivete la funzione `distance` che calcola la distanza tra due punti (`x1, y1`) e (`x2, y2`) nel piano (ricordiamo che la distanza fra due punti può essere calcolata per mezzo di un'applicazione del teorema di Pitagora). Tutti i numeri del problema devono essere di tipo `float`.

3.49 Che cosa fa questo programma?

```
1  // ex3.49.cpp
2  #include <iostream.h>
3
4  int main()
5  {
6     int c;
7
8     if ( ( c = cin.get() ) != EOF ) {
9        main();
10       cout << c;
11    }
12
13    return 0;
14 }
```

3.50 Che cosa fa questo programma?

```
1  // ex03_50.cpp
2  #include <iostream.h>
3
4  int mystery( int, int );
5
6  int main()
7  {
8     int x, y;
9
10    cout << "Enter two integers: ";
11    cin >> x >> y;
12    cout << "The result is " << mystery( x, y ) << endl;
13    return 0;
14 }
15
16 // Il parametro b dev'essere positivo
17 // per prevenire la ricorsione infinita
18 int mystery( int a, int b )
19 {
20    if ( b == 1 )
21       return a;
22    else
23       return a + mystery( a, b - 1 );
24 }
```


3.51 Dopo aver determinato cosa fa il programma dell'Esercizio 3.50, fatelo funzionare anche senza la restrizione che il secondo argomento sia non negativo.

3.52 Scrivete un programma che fa uso di un gran numero di funzioni matematiche di libreria. L'elenco delle funzioni è in Figura 3.2. Utilizzate queste funzioni e osservate i valori restituiti, organizzandoli in tabelle.

3.53 Trovate l'errore in ciascuno dei seguenti segmenti di programma e spiegate come correggerlo:

a)
```
float cube( float );   /* prototipo di funzione */
...
cube( float number )   /* definizione della funzione */
{
   return number * number * number;
}
```

b)
```
register auto int x = 7;
```

c)
```
int randomNumber = srand();
```

d)
```
float y = 123.45678;
int x;

x = y;
cout << static_cast< float>( x ) << endl;
```

e)
```
double square( double number )
{
   double number;
   return number * number;
}
```

f)
```
int sum( int n )
{
   if ( n == 0 )
      return 0;
   else
      return n + sum( n );
}
```

3.54 Modificate il programma in Figura 3.10 inserendo un meccanismo per le scommesse secondo quanto descritto di seguito. Impacchettate in una funzione la porzione di programma che fa una giocata completa. Inizializzate la variabile `bankBalance` a 1000 dollari. Chiedete all'utente di immettere la puntata. Controllate con un ciclo `while` che la scommessa sia minore o uguale a `bankBalance`: se non è così richiedete all'utente di immettere una puntata valida. Dopo aver ricevuto la puntata, eseguite una giocata. Se il giocatore vince, incrementate `bankBalance` del valore della puntata e visualizzate il suo nuovo valore. Se il giocatore perde, decrementate `bankBalance` del valore della puntata, visualizzate il nuovo valore di `bankBalance`, controllate se `bankBalance` è sceso a zero, e se è così visualizzate il messaggio `"Mi dispiace, sei in bancarotta!"`. Man mano che il gioco va avanti visualizzate vari messaggi, come `"Oh, stiamo rischiando grosso, eh?"` oppure `"Andiamo, riprova ancora!"`, o ancora `"Stiamo andando forte! È ora di incassare!"`

3.55 Scrivete un programma in C++ che utilizza la funzione inline `circleArea` per chiedere all'utente il raggio di un cerchio, per calcolare e visualizzare l'area del cerchio.

3.56 Scrivete un programma completo delle due funzioni alternative indicate di seguito, ognuna delle quali triplica la variabile `count` definita in `main`. Confrontate i due tipi di approccio al problema. Le due funzioni sono:

a) La funzione `tripleCallByValue` che passa una copia di `count` in una chiamata per valore, triplica la copia e restituisce un nuovo valore.

b) La funzione `tripleByReference` che passa `count` con una chiamata per riferimento passando come parametro un riferimento e triplica la copia originale di `count` tramite l'alias (cioè il riferimento passato come parametro).

3.57 Qual è lo scopo dell'operatore unario di risoluzione dello scope?

3.58 Scrivete un programma che utilizza una funzione generica denominata `min` per determinare il minore tra due argomenti. Verificate il programma utilizzando dei valori interi, dei caratteri e dei numeri a virgola mobile.

3.59 Scrivete un programma che utilizza una funzione generica denominata `max` per determinare il maggiore tra due argomenti. Verificate il programma utilizzando dei valori interi, dei carattere e dei numeri a virgola mobile.

3.60 Determinate se i segmenti di programma che vi proponiamo contengono errori. Per ogni errore trovato, *spiegate come correggerlo. Nota: è possibile che in un segmento non ci sia alcun errore.*

a)
```
template < class A >
int sum( int num1, int num2, int num3 )
{
   return num1 + num2 + num3;
}
```

b)
```
void printResults( int x, int y )
{
  cout << "The sum is " << x + y << '\n';
   return x + y;
}
```

c)
```
template < A >
A product( A num1, A num2, A num3 )
{
   return num1 * num2 * num3;
}
```

d)
```
double cube( int );
int cube( int );
```

CAPITOLO 4

Gli array

Obiettivi

- Comprendere le strutture dati di tipo array
- Comprendere come utilizzare gli array per memorizzare, ordinare ed effettuare ricerche su liste e tabelle di valori
- Comprendere come dichiarare, inizializzare e riferirsi agli (o referenziare gli) elementi di un array
- Comprendere come passare parametri di tipo array alle funzioni
- Comprendere i concetti fondamentali degli algoritmi di ordinamento
- Imparare a utilizzare gli array multidimensionali

4.1 Introduzione

In questo capitolo si introducono le strutture dati: un argomento teorico molto importante. Gli *array* sono un tipo di struttura dati composta da più elementi dello stesso tipo correlati tra di loro. Nel Capitolo 6 parleremo delle *strutture* e delle *classi*: anch'esse servono a memorizzare elementi correlati, ma i tipi di dato possono essere anche diversi. Gli array e le strutture sono entità statiche: essi mantengono le stesse dimensioni dall'inizio alla fine del programma. Naturalmente anch'esse possono esistere soltanto per un tempo limitato durante il programma: infatti se appartengono alla categoria di memorizzazione automatica saranno create in un dato blocco e distrutte quando tale blocco termina. Nel Capitolo 4 del volume Tecniche Avanzate introdurremo le strutture dati dinamiche, come liste, code, pile e alberi: queste hanno la capacità di espandersi e ridimensionarsi durante l'esecuzione del programma. *In questo capitolo faremo uso dello stile C per gli array basati sui* puntatori (studieremo i puntatori nel Capitolo 5). Nel Capitolo 8 e in una sezione finale di questo testo dedicata alla libreria standard torneremo a parlare degli array, alla luce delle nuove conoscenze che avremo acquisito riguardo la programmazione orientata agli oggetti. Scopriremo che gli array "basati sugli oggetti" sono più sicuri e versatili di quelli che studieremo in questo capitolo, ovvero degli array in vecchio stile C basati sui puntatori.

4.2 Gli array

Un array è un gruppo di locazioni di memoria consecutive, ognuna delle quali ha lo stesso nome ed è dello stesso tipo di dato. Per riferirci ad una locazione particolare, ovvero un elemento dell'array, si deve specificare il nome dell'array e il *numero posizionale* dell'elemento desiderato.

La Figura 4.1 mostra l'array di interi `c`. Questo array contiene dodici *elementi*: per riferirci ad uno di essi occorre scrivere il nome dell'array e il numero posizionale dell'elemento racchiuso tra parentesi quadre (`[]`). Il primo elemento di ogni array è sempre l'*elemento di posizione zero*. Perciò il primo elemento di `c` è `c[0]`, il secondo elemento è `c[1]`, il settimo elemento è `c[6]`, e in generale l'elemento *i*-esimo dell'array `c` è `c[i - 1]`. I nomi degli array seguono le stesse convenzioni dei nomi delle altre variabili.

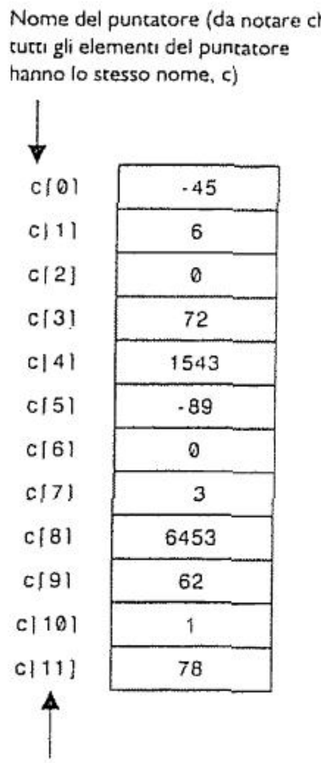


Figura 4.1 Un array di 12 elementi.

Il numero posizionale racchiuso tra parentesi quadre viene detto *indice*. Un indice può essere un intero o un'espressione intera. Se come indice si ha un'espressione, il programma calcola prima il valore di tale espressione, per poter determinare poi l'elemento dell'array da prendere in considerazione. Supponiamo, per esempio, che la variabile `a` sia uguale a 5 e che la variabile `b` sia uguale a 6. L'istruzione

```
c[a + b] += 2;
```

incrementa di 2 l'elemento `c[11]`. Osservate che il nome di un array completo di indice è un *lvalue*, cioè può essere utilizzato nella parte sinistra di un assegnamento.

Esaminiamo ora più da vicino l'array `c` in Figura 4.1. Il *nome* dell'intero array è `c`. I nomi dei suoi dodici elementi sono `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`. Il *valore* di `c[0]` è `-45`, il valore di `c[1]` è `6`, il valore di `c[2]` è `0`, il valore di `c[7]` è `62` e il valore di `c[11]` è `78`. Per visualizzare la somma dei valori dei primi tre elementi dell'array `c` scriviamo l'istruzione

```
cout << c[0]+c[1] + c[2] << endl;
```

Se vogliamo dividere per `2` il valore del settimo elemento di `c` e assegnare il risultato alla variabile `x`, possiamo scrivere invece

```
x = c[6] / 2;
```

Errore tipico 4.1

È importante notare la differenza tra le due locuzioni "settimo elemento di un array" e "elemento sette di un array". Dato che gli indici iniziano da 0, il "settimo elemento di un array" ha indice 6, mentre un "elemento sette di un array", che ha appunto indice 7, si riferisce in realtà all'ottavo elemento dell'array. Ciò può generare un po' di confusione e causare errori del tipo "fuori di uno" (in inglese off-by-one), cioè l'accesso all'elemento successivo all'ultimo.

Le parentesi quadre che racchiudono l'indice costituiscono un operatore del C++. L'ordine di precedenza delle parentesi quadre è uguale a quello delle parentesi tonde.

Lo schema in Figura 4.2 illustra la precedenza e l'associatività degli operatori che abbiamo studiato finora. Essi sono elencati in ordine decrescente di precedenza, insieme con la loro associatività e il loro tipo.

Operatori	Associatività	Tipo
`()` `[]`	sx a dx	più alto
`++` `--` `+` `-` `!` `static_cast<type>()`	dx a sx	unario
`*` `/` `%`	sx a dx	moltiplicativo
`+` `-`	sx a dx	additivo
`<<` `>>`	sx a dx	inserzione/estrazione
`<` `<=` `>` `>=`	sx a dx	relazionale
`==` `!=`	sx a dx	d'uguaglianza
`&&`	sx a dx	AND logico
`||`	sx a dx	OR logico
`?:`	dx a sx	condizionale
`=` `+=` `-=` `*=` `/=` `%=`	dx a sx	assegnamento
`,`	sx a dx	virgola

Figura 4.2 Precedenza e associatività degli operatori.

4.3 Come si dichiara un array

Gli array occupano spazio in memoria: per ogni array occorre quindi specificare il tipo e il numero di elementi che esso contiene, in modo che il compilatore possa riservargli una porzione di memoria di dimensioni adeguate.

Se vogliamo indicare al compilatore di riservare spazio per i 12 elementi dell'array di interi `c`, possiamo utilizzare la dichiarazione

```
int c[ 12 ];
```

In una sola istruzione potete dichiarare più di un array. Nella seguente, per esempio, ne dichiariamo due, `b` di 100 elementi interi e `x` di 27 elementi interi:

```
int b[ 100 ], x[ 27 ];
```

Gli array possono contenere anche altri tipi di dato, non solo gli interi, con il solo vincolo che questi siano dello stesso tipo. Per esempio, potete utilizzare un array di `char` per memorizzare una stringa di caratteri.

Nel Capitolo 5 parleremo delle stringhe e indicheremo i punti in comune che hanno con gli array e la stretta correlazione che esiste tra puntatori e array. La maggior parte di queste somiglianze è un'eredità del buon vecchio C. Più avanti in questo libro torneremo sulle stringhe, vedendolecome oggetti a tutti gli effetti.

4.4 Alcuni esempi di array

Il programma in Figura 4.3 inizializza e visualizza in formato tabulare gli elementi dell'array n di dieci elementi interi; e per far questo fa uso di un ciclo `for`. La prima istruzione di output visualizza l'intestazione delle colonne in cui `for` scriverà i valori; come ricorderete, `setw` specifica la larghezza del campo per il successivo valore da inviare in output.

```
1   // f 4.3: fig04_03.cpp
2   // Inizializzazione di un array
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      int i, n[ 10 ];
9
10     for ( i = 0; i < 10; i++ )        // Inizializza l'array
11        n[ i ] = 0;
12
13     cout << "Element" << setw( 13 ) << "Value" << endl;
14
15     for ( i = 0; i < 10; i++ )        // Visualizza l'array
16        cout << setw(7) << i << setw(13) << n[i] << endl;
17
18     return 0;
19  }
```

Figura 4.3 Inizializzazione a zero degli elementi di un array (continua)


```
Element       Value
      0           0
      1           0
      2           0
      3           0
      4           0
      5           0
      6           0
      7           0
      8           0
      9           0
```

Figura 4.3 Inizializzazione a zero degli elementi di un array.

Gli elementi di un array possono essere anche inizializzati direttamente durante la dichiarazione dell'array: per fare ciò basta specificare un segno di uguaglianza seguito da una lista di valori separati da virgole e racchiusi tra parentesi graffe: questi valori prendono il nome di *inizializzatori* o *valori di inizializzazione*. Il programma in Figura 4.4 inizializza un array di interi con dieci inizializzatori e lo visualizza successivamente in formato tabulare.

```
1   // Fig. 4.4: fig04_04.cpp
2   // Inizializzazione di un array nella dichiarazione
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      int n[10] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
9
10     cout << "Element" << setw( 13 ) << "Value" << endl;
11
12     for ( int i = 0; i < 10; i++ )
13        cout << setw(7) << i << setw(13) << n[ i ] << endl;
14
15     return 0;
16  }
```

```
Element       Value
      0          32
      1          27
      2          64
      3          18
      4          95
      5          14
      6          90
      7          70
      8          60
      9          37
```

Figura 4.4 Inizializzazione degli elementi di un array durante la dichiarazione.

Se gli inizializzatori sono meno degli elementi dell'array, i restanti elementi vengono automaticamente azzerati. Per esempio, questa istruzione azzera tutti gli elementi dell'array n in Figura 4.3:

```
int n[ 10 ] = { 0 };
```

L'unico elemento inizializzato esplicitamente è il primo, mentre gli altri sono azzerati automaticamente, perché ci sono meno inizializzatori che elementi. Ricordate però che non sempre gli elementi di un array vengono azzerati in modo automatico; nel caso della dichiarazione senza inizializzazione, infatti, l'array contiene inizialmente dei valori indefiniti. Il programmatore deve inizializzarne almeno un elemento, cioè il primo, perche venga azzerato tutto l'array. Il metodo in Figura 4.3 potrà essere utilizzato ovunque nei vostri programmi.

Errore tipico 4.2

Se occorre inizializzare gli elementi di un array ma dimenticate di farlo, commettete un errore di logica.

La seguente dichiarazione di array

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

darà un errore di sintassi, perché ci sono 6 inizializzatori per 5 elementi.

Errore tipico 4.3

Se fornite più inizializzatori del numero di elementi effettivamente presenti nell'array, commettete un errore di sintassi.

Potete dichiarare un array e ometterne la dimensione, se fornite contestualmente una lista di inizializzatori: il compilatore assumerà che il numero di elementi dell'array sia uguale al numero di inizializzatori. Per esempio:

```
int n[] = { 1, 2, 3, 4, 5 };
```

crea un array di cinque elementi.

Obiettivo efficienza 4.1

Se inizializzate un array durante la dichiarazione (in fase di compilazione) anziché con un assegnamento per ciascun elemento (in fase di esecuzione), il vostro programma sarà eseguito più velocemente.

Il programma in Figura 4.5 inizializza e visualizza in formato tabulare gli elementi dell'array di dieci interi s. I valori di inizializzazione sono generati moltiplicando di volta in volta il contatore del ciclo per 2 e sommando 2.

```
1   // Fig. 4.5: fig04_05.cpp
2   // Inizializzazione dell'array s con i pari tra 2 e 20.
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
```

Figura 4.5 Generazione di valori da porre negli elementi di un array (continua)


```
8      const int arraySize = 10;
9      int j, s[ arraySize ];
10
11     for ( j = 0; j < arraySize; j++ )   // imposta i valori
12        s[ j ] = 2 + 2 * j;
13
14     cout << "Element" << setw( 13 ) << "Value" << endl;
15
16     for (j = 0; j < arraySize; i++)   // visualizza i valori
17        cout << setw(7) << j << setw(13) << s[ j ] << endl;
18
19     return 0;
20  }
```

```
Element      Value
      0          2
      1          4
      2          6
      3          8
      4         10
      5         12
      6         14
      7         16
      8         18
      9         20
```

Figura 4.5 Generazione di valori da porre negli elementi di un array.

La linea 8

```
const int arraySize = 10;
```

utilizza il qualificatore `const` per dichiarare una *variabile costante* o *variabile a sola lettura*, con valore `10`. Le variabili costanti devono essere inizializzate da un'espressione costante durante la dichiarazione, e da quel momento non è più possibile modificarle (Figura 4.6 e Figura 4.7). *Le variabili a sola lettura sono anche dette costanti nominali.* Se ci fate caso, la locuzione "variabile costante" è un ossimoro, cioè una contraddizione in termini.

```
1   // Fig. 4.6: fig04_06.cpp
2   // Utilizzo di una variabile a sola lettura inizializzata
3   // correttamente
4   #include <iostream.h>
5
6   int main()
7   {
8      const int x = 7;   // variabile a sola lettura
9                         // inizializzata
```

Figura 4.6 Come si inizializza e si utilizza correttamente una variabile a sola lettura (continua).

```
10
11       cout << "The value of constant variable x is: "
12            << x << endl;
13
14       return 0;
15    }
```

```
The value of constant variable x is: 7
```

Figura 4.6 Come si inizializza e si utilizza correttamente una variabile a sola lettura

```
1    // Fig. 4.7: fig04_07.cpp
2    // Un oggetto const deve essere inizializzato
3
4    int main()
5    {
6       const int x; // Errore: x deve essere inizializzato
7
8       x = 7;       // Errore: non è possibile modificare una
9                    // variabile a sola lettura
10      return 0;
11   }
```

```
Compiling FIG04_7.CPP:
Error FIG04_7.CPP 6: Constant variable 'x' must be
initialized
Error FIG04_7.CPP 8: Cannot modify a const object
```

Figura 4.7 Un oggetto const deve essere inizializzato.

Errore tipico 4.4

Se assegnate un valore a una variabile a sola lettura nel corso del programma, commettete un errore di sintassi.

Le variabili costanti possono essere utilizzate ovunque siano consentite espressioni costanti. In Figura 4.5 utilizziamo la variabile a sola lettura `arraySize` per specificare la dimensione dell'array `s` nella dichiarazione

```
int j, s[ arraySize ];
```

Errore tipico 4.5

Potete utilizzare solo costanti nella dichiarazione di array statici e automatici. Se non utilizzate un valore costante commettete un errore di sintassi.

L'uso delle variabili costanti rende i programmi più facilmente adattabili a nuovi problemi, cioè *scalabili*. Nel programma in Figura 4.5 potreste modificare il ciclo `for` in modo che riempia un array di 1000 elementi, anziché di 10, cambiando semplicemente il valore di `arraySize` all'inizio del programma. Se non avessimo utilizzato la variabile a sola lettura `arraySize` avremmo dovuto cambiare il programma in tre punti diversi per ottenere lo stesso risultato. Se lavorate a un progetto di grandi dimensioni tenete conto di questa utile tecnica.

Ingegneria del software 4.1

Se utilizzate variabili costanti anziché costanti numeriche per definire le dimensioni degli array, scriverete programmi più scalabili.

Buona abitudine 4.1

Se definite la dimensione di un array con una variabile a sola lettura anziché con una costante numerica scriverete un programma più leggibile ed inoltre questa tecnica vi consente di sbarazzarvi dei cosiddetti "numeri magici". Supponiamo di aver definito un array di dimensione 10: in gran parte delle istruzioni che operano sull'array sarà presente inevitabilmente il numero 10. Chi legge il vostro programma potrà confondere il valore 10, che si riferisce alla dimensione dell'array, con altri 10 sparsi nel programma ma che non si riferiscono in alcun modo all'array e hanno quindi un significato del tutto diverso.

Il programma in Figura 4.8 somma i valori dei dodici elementi dell'array `a`. L'istruzione nel corpo di `for` calcola il totale. Vi facciamo notare, per inciso, che se non avessimo fornito gli inizializzatori di `a`, tutti i valori dell'array dovrebbero essere richiesti probabilmente all'utente durante l'esecuzione. Il seguente costrutto `for`

```
for ( int j = 0; j < arraySize; j++ )
   cin >> a[ j ];
```

legge un valore alla volta dalla tastiera e lo memorizza nell'elemento `a[ j ]`.

```
1    // Fig. 4.8: fig04_08.cpp
2    // Calcolo della somma degli elementi di un array
3    #include <iostream.h>
4
5    int main()
6    {
7       const int arraySize = 12;
8       int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99,
9                              16, 45, 67, 89, 45 };
10      int total = 0;
11
12      for ( int i = 0; i < arraySize; i++ )
13         total += a[ i ];
14
15      cout << "Total of array element values is " << total
16           << endl;
17
18      return 0;
19   }
```

```
Total of array element values is 383
```

Figura 4.8 Calcolo della somma degli elementi di un array.

Il nostro prossimo esempio riepiloga i dati di un sondaggio d'opinione in un array. Ecco la definizione del nostro problema:

Abbiamo chiesto a quaranta studenti di esprimere un giudizio sulla qualità di un locale pubblico frequentato prevalentemente da giovani con un voto da 1 a 8. Trasferite le 40 risposte in un array di interi e visualizzate un riepilogo dei risultati.

Applicazioni di questo genere costituiscono un uso tipico degli array (cfr. Figura 4.9). Vogliamo dunque stilare un riepilogo del numero di risposte di ogni tipo: quanti sono gli 1, quanti sono i 2 e così via. L'array `responses` contiene 40 elementi, le risposte degli studenti. Utilizziamo l'array `frequency`, composto da 9 elementi, per contare le occorrenze di ogni risposta. Abbiamo scelto di ignorare il primo elemento `frequency[ 0 ]` perché è più logico mettere in relazione ogni indice con il tipo di risposta corrispondente, memorizzando in `frequency[ 1 ]` le occorrenze degli 1 e così via. Grazie a questo stratagemma potremo anche utilizzare direttamente la risposta come indice di `frequency`.

```
1   // Fig. 4.9: fig04_09.cpp
2   // programm per l'analisi di un sondaggio effettuato su studenti
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      const int responseSize = 40, frequencySize = 9;
9      int responses[responseSize]={1, 2, 6, 4, 8, 5, 7, 8, 7
10     1, 6, 3, 8, 6, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 7,
11     5, 6, 6, 7, 5, 6, 7, 5, 6, 4, 8, 6, 8, 7 };
12     int frequency[ frequencySize ] = { 0 };
13
14     for ( int answer = 0; answer < responseSize; answer++ )
15        ++frequency[ responses[ answer ] ];
16
17     cout << "Rating" << setw( 17 ) << "Frequency" << endl;
18
19     for ( int rating = 1; rating < frequencySize; rating++ )
20        cout << setw( 6 ) << rating
21             << setw( 17 ) << frequency[ rating ] << endl;
22
23     return 0;
24  }
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                9
     8                7
```

Figura 4.9 Programma per analizzare un sondaggio effettuato su studenti.

Buona abitudine 4.2

Sforzatevi sempre di scrivere programmi chiari e leggibili. A volte conviene sacrificare memoria o tempo di elaborazione per evitare che un programma risulti oscuro quando viene riletto a distanza di un po' di tempo.

Obiettivo efficienza 4.2

Ci sono casi in cui occorre trascurare l'aspetto della chiarezza per favorire unicamente l'aspetto delle prestazioni.

Il primo ciclo `for` riceve le risposte una alla volta dall'array `responses` e incrementa *uno dei dieci contatori dell'array* `frequency`, da `frequency[ 1 ]` a `frequency[ 10 ]`. L'istruzione più importante del ciclo è

```
++frequency[ responses[ answer ] ];
```

Questa istruzione incrementa il contatore appropriato di `frequency`, a seconda del valore di `responses[ answer ]`. Per esempio, se il contatore `answer` vale `0`, il valore di `responses[ answer ]` è `1`, per cui l'istruzione `++frequency[ responses[ answer ] ];` viene interpretata come

```
++frequency[ 1 ];
```

che incrementa l'elemento numero 1 dell'array. Se `answer` è `1`, il valore di `responses[ answer ]` è `2`, per cui l'istruzione `++frequency[ responses[ answer ] ];` viene interpretata come

```
++frequency[ 2 ];
```

che incrementa l'elemento numero 2 dell'array. Se `answer` è `2`, il valore di `responses[ answer ]` è `6`, per cui l'istruzione `++frequency[ responses[ answer ] ];` viene interpretata come

```
++frequency[ 6 ];
```

che incrementa l'elemento numero 6, e così via. Osservate che, indipendentemente dal numero di persone intervistate, abbiamo bisogno di 11 elementi soltanto, incluso l'elemento zero che abbiamo scartato, per stilare il nostro riepilogo finale. Se i dati contenessero valori sbagliati, per esempio 13, il programma cercherebbe di incrementare di 1 il valore di `frequency[ 13 ]`. Questa istruzione oltrepassa i limiti dell'array. *Il C++ non effettua alcun controllo sulle istruzioni per evitare che oltrepassino i limiti di un array, accedendo a un elemento che non esiste.* Questo significa che un'istruzione può oltrepassare i limiti di un array senza che il sistema vi avverta con un messaggio, e può farlo nei due sensi, oltre l'ultimo elemento e prima dell'elemento zero. Il programmatore deve controllare autonomamente che tutti i riferimenti ad un array rimangano in limiti significativi. Tuttavia il C++ è un linguaggio estensibile: nel Capitolo 8 vedremo come implementare un array come tipo definito dall'utente grazie alle classi. La nostra nuova definizione di array ci consentirà di effettuare operazioni che sono impensabili con gli array tradizionali del C++. Per esempio potremo confrontare due array direttamente, assegnare un intero array a un altro, effettuare l'input e l'output con `cin` e `cout`, inizializzare automaticamente un array, evitare il problema dell'accesso a elementi oltre i limiti dell'array e persino cambiare l'intervallo di indici da utilizzare (e il loro tipo), in modo che il primo elemento non debba avere necessariamente indice 0.

Errore tipico 4.6

Se referenziate un elemento che si trova oltre i limiti dell'array durante l'esecuzione del programma commettete un errore logico, ma non di sintassi.

Collaudo e messa a punto 4.1

Se in un ciclo operate su un array, controllate che l'indice non scenda mai al di sotto di 0 e non superi mai il numero di elementi dell'array meno uno. Assicuratevi che la condizione di termine del ciclo non acceda a elementi che non esistono.

Collaudo e messa a punto 4.2

I programmi dovrebbero verificare la validità di tutti i valori accettati in input, in modo da evitare calcoli che si basano su valori che sono errati sin dall'inizio e che quindi restituiranno risultati senza significato.

Obiettivo portabilità 4.1

Se accedete a elementi di un array che si trovano oltre i suoi limiti, ogni sistema reagirà in modo diverso: premettiamo, però, che normalmente la reazione sarà disastrosa.

Collaudo e messa a punto 4.3

Quando inizieremo a studiare le classi (Capitolo 6) vedremo come sviluppare un "array intelligente", che controlla automaticamente se gli indici sono validi durante l'esecuzione. Tipi di dato di questo genere sono utili perché riducono il numero di errori che possono passare inosservati.

Nel nostro prossimo esempio (Figura 4.10) leggiamo dei numeri da un array e creiamo un grafico a barre, o istogramma, sulla base dei loro valori. A ogni numero visualizzato accodiamo una barra di tanti asterischi quanto è il valore del numero.

Il ciclo `for` nidificato è responsabile del "tracciamento" dell'istogramma. Notate l'uso di `endl` per terminare ogni barra.

Errore tipico 4.7

Anche se è possibile utilizzare lo stesso contatore in due cicli `for` nidificati, ciò normalmente comporta un errore logico.

```
1   // Fig. 4.10: fig04_10.cpp
2   // Programma che visualizza un istogramma
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      const int arraySize = 10;
9      int n[arraySize] = {19, 3, 15, 7, 11, 9, 13, 5, 17, 1};
10
11     cout << "Element" << setw( 13 ) << "Value"
12          << setw( 17 ) << "Histogram" << endl;
13
```

Figura 4.10 Programma che visualizza un istogramma (continua)


```
14     for ( int i = 0; i < arraySize ; i++ ) {
15        cout << setw( 7 ) << i << setw( 13 )
16             << n[ i ] << setw( 9 );
17
18        for (int j=0; j < n[i]; j++)    // visualizza una barra
19           cout << '*';
20
21        cout << endl;
22     }
23
24     return 0;
25  }
```

```
Element        Value      Histogram
      0           19      *******************
      1            3      ***
      2           15      ***************
      3            7      *******
      4           11      ***********
      5            9      *********
      6           13      *************
      7            5      *****
      8           17      *****************
      9            1      *
```

Figura 4.10 Programma che visualizza un istogramma.

Collaudo e messa a punto 4.4

Anche se è possibile modificare il contatore nel corpo di `for`, è bene evitare di farlo, perché si possono commettere errori logici difficili da individuare.

Nel Capitolo 3 vi abbiamo promesso che avremmo mostrato un metodo più elegante per scrivere il programma del lancio dei dadi in Figura 3.8. In quel programma dovevamo tirare un dado a sei facce 6000 volte e controllare con un'istruzione switch che numero fosse uscito per verificare se il generatore di numeri casuali del computer funzionasse correttamente. La nuova versione del programma,presentato in Figura 4.11, utilizza gli array.

```
1   // Fig. 4.11: fig04_11.cpp
2   // Programma che lancia un dado 6000 volte
3   #include <iostream.h>
4   #include <iomanip.h>
5   #include <stdlib.h>
6   #include <time.h>
7
8   int main()
9   {
10     const int arraySize = 7;
```

Figura 4.11 Programma di lancio dei dadi che utilizza gli array anziché lo switch (continua)

```
11    int face, frequency[ arraySize ] = { 0 };
12
13    srand( time( 0 ) );
14
15    for ( int roll = 1; roll <= 6000; roll++ )
16       ++frequency[ 1 + rand() % 6 ]; // sostituisce lo
17                                      // switch di linea 20 di Figura 3.8
18
19    cout << "Face" << setw( 13 ) << "Frequency" << endl;
20
21    // ignora l'elemento 0 nell'array frequency
22    for ( face = 1; face < arraySize ; face++ )
23       cout << setw( 4 ) << face
24            << setw( 13 ) << frequency[ face ] << endl;
25
26    return 0;
27 }
```

```
Face        Frequency
   1             1037
   2              987
   3             1013
   4             1028
   5              952
   6              983
```

Figura 4.11 Programma di lancio dei dadi che utilizza gli array anziché lo switch.

Finora abbiamo incontrato soltanto array di interi. Vi ricordiamo che gli array possono contenere dati di qualunque tipo. Adesso vedremo come memorizzare una stringa di caratteri in un array di caratteri. Fino a questo momento l'unico utilizzo che abbiamo fatto delle stringhe è stata la visualizzazione con `cout` e l'operatore `<<` Una stringa come `"hello"` è concettualmente una sequenza di caratteri, e quindi si presta bene a essere rappresentata come array. Gli array di caratteri hanno però alcune caratteristiche peculiari.

Un array di caratteri può essere inizializzato da una stringa literal. Per esempio, la dichiarazione

```
char string1[] = "first";
```

inizializza gli elementi dell'array `string1` con i caratteri individuali della stringa literal `"first"` La dimensione di `string1` viene determinata dal compilatore sulla base della lunghezza del literal. È importante sapere che la stringa `"first"` contiene cinque caratteri *più* un carattere speciale che segnala la fine della stringa, il *carattere nullo*. Ciò significa che `string1` contiene in realtà sei elementi, non cinque. La rappresentazione costante del carattere nullo è `'\0'`, una barra rovesciata seguita da zero. Tutte le stringhe terminano con questo carattere, per cui se un array di caratteri deve rappresentare una stringa, fate in modo che sia abbastanza capiente per contenerne tutti i caratteri più il carattere nullo.

Gli array di caratteri possono anche essere inizializzati con ciascuna delle costanti carattere che compongono una stringa. Quello che intendiamo è che la precedente dichiarazione può essere sostituita dalla seguente, decisamente più tediosa:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Dato che una stringa è in realtà un array di caratteri, possiamo anche accedere ai caratteri individuali tramite gli indici. Per esempio `string1[0]` è il carattere `'f'` e `string1[3]` è il carattere `'s'`

Possiamo prendere una stringa dall'input e memorizzarla direttamente in un array di caratteri, grazie a `cin` e `>>` Per esempio, la dichiarazione

```
char string2[ 20 ];
```

crea un array di caratteri in grado di memorizzare al massimo una stringa di 19 caratteri più il carattere nullo che la termina (20 caratteri in totale). L'istruzione

```
cin >> string2;
```

legge una stringa dalla tastiera e la memorizza in `string2`. Come avrete notato, nell'istruzione precedente abbiamo fornito soltanto il nome dell'array: non è presente alcuna informazione sulla sua dimensione. Sta al programmatore assicurarsi che l'array sia sufficientemente grande per memorizzare la stringa più lunga che può venire dall'input. `cin` legge i caratteri dalla tastiera uno alla volta finché non viene incontrato un carattere di spaziatura, senza curarsi delle dimensioni dell'array. Può accadere quindi che `cin` legga più caratteri di quelli che è possibile conservare in `string2`.

Errore tipico 4.8

Se a `cin` non viene fornito un array di caratteri abbastanza grande per memorizzare anche le stringhe più lunghe possibili in input, potreste perdere dei dati o incorrere in altri gravi errori durante l'esecuzione del programma.

Con `cout` potete visualizzare un array di caratteri che contiene una stringa terminata dal carattere nullo. L'array `string2` può essere visualizzato con l'istruzione

```
cout << string2 << endl;
```

Notate che `cout <<`, proprio come `cin >>`, non si cura della dimensione dell'array di caratteri. I caratteri della stringa sono inviati nell'output finché non viene raggiunto il carattere nullo. Il programma in Figura 4.12 inizializza un array di caratteri tramite una stringa literal, poi legge una stringa e la memorizza nell'array, quindi visualizza l'array come stringa e infine accede ai singoli caratteri che la compongono.

```
 1  // Fig. 4_12: fig04_12.cpp
 2  // Array di caratteri come stringhe
 3  #include <iostream.h>
 4
 5  int main()
 6  {
 7     char string1[ 20 ], string2[] = "string literal";
 8
 9     cout << "Enter a string: ";
10     cin >> string1;
```

Figura 4.12 Trattamento di array di caratteri come stringhe (continua)

```
11      cout << "string1 is: " << string1
12           << "\nstring2 is: " << string2
13           << "string1 with spaces between characters is:\n";
14
15      for ( int i = 0; string1[ i ] != '\0'; i++ )
16         cout << string1[ i ] << ' ';
17
18      cin >> string1;  // legge "there"
19      cout << "\nstring1 is: " << string1 << endl;
20
21      cout << endl;
22      return 0;
23   }
```

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
string1 is: there
```

Figura 4.12 *Trattamento di array di caratteri come stringhe.*

La Figura 4.12 utilizza un ciclo `for` per accedere a tutti i caratteri della stringa `string1` e inviarli in output separandoli con degli spazi. La condizione del ciclo `for`, `string1[ i ] != '\0'`, è vera *finché non viene incontrato il carattere nullo.*

Nel Capitolo 3 abbiamo parlato della categoria di memorizzazione `static`. Una variabile locale di tipo `static` di una funzione esiste per tutta la durata del programma, ma è visibile solo durante l'esecuzione della funzione.

Obiettivo efficienza 4.3

Se dichiarate un array locale come `static`*, esso non verrà creato, inizializzato e distrutto a ogni chiamata della funzione. In questo modo migliorerà l'efficienza del programma.*

Gli array dichiarati come `static` sono inizializzati una sola volta all'inizio dell'esecuzione. *Se non ci pensa il programmatore, lo fa il compilatore: inizializzando automaticamente* tutti gli elementi.

In Figura 4.13 vedete la funzione `staticArrayInit`, che dichiara un array locale `static`, e la *funzione* `automaticArrayInit`, *che dichiara un array locale automatico.* La funzione `staticArrayInit` viene chiamata due volte. L'array statico è inizializzato a zero dal compilatore.

La *funzione visualizza l'array, incrementa di 5 ciascun elemento e infine visualizza* nuovamente l'array. Alla seconda chiamata della funzione l'array statico conserva ancora i valori memorizzati durante la prima chiamata. Anche `automaticArrayInit` viene chiamata due volte. Gli elementi dell'array locale automatico sono inizializzati con i valori 1, 2 e 3. La funzione visualizza l'array, incrementa di 5 ciascun elemento e infine visualizza nuovamente l'array.

La seconda volta che la funzione viene chiamata, gli elementi dell'array verranno inizializzati nuovamente con i valori 1, 2 e 3, perché l'array è stato creato di nuovo.

```
1   // Fig. 4.13: fig04_13.cpp
2   // Gli array static sono inizializzati a zero
3   #include <iostream.h>
4
5   void staticArrayInit( void );
6   void automaticArrayInit( void );
7
8   int main()
9   {
10     cout << "First call to each function:\n";
11     staticArrayInit();
12     automaticArrayInit();
13
14     cout << "\n\nSecond call to each function:\n";
15     staticArrayInit();
16     automaticArrayInit();
17     cout << endl;
18
19     return 0;
20  }
21
22  // funzione che mostra l'uso di un array locale statico
23  void staticArrayInit( void )
24  {
25     static int array1[ 3 ];
26     int i;
27
28     cout << "\nValues on entering staticArrayInit:\n";
29
30     for ( i = 0; i < 3; i++ )
31        cout <<"array1[" << i << "]=" << array1[ i ] << "  ";
32
33     cout << "\nValues on exiting staticArrayInit:\n";
34
35     for ( i = 0; i < 3; i++ )
36        cout << "array1[" << i << "] = "
37             << ( array1[ i ] += 5 ) << "  ";
38  }
39
40  // funzione che mostra l'uso di un array locale automatico
41  void automaticArrayInit( void )
42  {
43     int i, array2[ 3 ] = { 1, 2, 3 };
44
45     cout << "\n\nValues on entering automaticArrayInit:\n";
```

Figura 4.13 Confronto tra inizializzazione di array statici e array automatici (continua)

```
46
47       for ( i = 0; i < 3; i++ )
48          cout <<"array2[" << i << "]=" << array2[ i ] << "   ";
49
50       cout << "\nValues on exiting automaticArrayInit:\n";
51
52       for ( i = 0; i < 3; i++ )
53          cout << "array2[" << i << "] = "
54               << ( array2[ i ] += 5 ) << "   ";
55    }
```

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:
Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Figura 4.13 Confronto tra inizializzazione di array statici e array automatici.

Errore tipico 4.9

Se assumete che gli elementi di un array locale `static` *siano azzerati ogni volta che la funzione viene chiamata, potreste commettere degli errori logici nel programma.*

4.5 Il passaggio di un array a una funzione

Per passare un array a una funzione basta specificarne il nome senza le parentesi quadre. Per esempio, se avete dichiarato l'array `hourlyTemperature` come

```
int hourlyTemperature[ 24 ];
```

la chiamata

```
modifyArray( hourlyTemperature, 24 );
```


passa l'array `hourlyTemperature` e la sua dimensione alla funzione `modifyArray`. Quando si passa un array a una funzione è buona regola passarle anche la sua dimensione, per informare la funzione del numero di elementi a cui accedere. Se non lo facciamo, dovremmo incorporare questa "conoscenza" all'interno della stessa funzione, o peggio dovremmo memorizzarlo in una variabile globale. Nel Capitolo 8 introdurremo la classe `Array`, e vedremo che un `Array` sarà in grado di "conoscere" in ogni momento la propria dimensione. Con un tipo di dato come questo non saremo più costretti a passare alle funzioni anche le dimensioni degli array.

Il C++ passa gli array alle funzioni forzando una chiamata per riferimento: ciò significa che le funzioni chiamate possono modificare il valore degli elementi nell'array originario. Il nome di un array è in realtà l'indirizzo del suo primo elemento. Passando a una funzione il nome di un array le passiamo in realtà l'indirizzo iniziale dell'array, e la funzione chiamata saprà con precisione dove si trova l'array in memoria. Ciò significa anche che se la funzione modifica un elemento dell'array, effettua la modifica direttamente sulle locazioni di memoria dell'array originario.

Obiettivo efficienza 4.4

L'utilizzo forzato della chiamata per riferimento nel passaggio di array ha senso per motivi di efficienza. Se gli array fossero passati per valore, dovrebbero essere prima trascritti interamente in una copia temporanea in memoria. Nel caso di grossi array l'operazione di copia comporterebbe un notevole spreco di memoria e di tempo di elaborazione.

Ingegneria del software 4.2

È possibile passare un array per valore con una tecnica che illustreremo nel Capitolo 6, ma si tratta di una situazione poco frequente.

Fate attenzione a distinguere le due cose: gli array interi sono passati per riferimento, mentre i singoli elementi di un array sono passati per valore, esattamente come qualsiasi variabile ordinaria, detta anche *scalare*. Per passare un elemento di un array a una funzione utilizzate il nome dell'array e l'indice dell'elemento. Nel Capitolo 5, mostreremo come effettuare la chiamata per riferimento degli scalari, come le variabili ordinarie e gli elementi di array.

Se una funzione riceve un array come argomento, deve indicarlo espressamente nella lista dei parametri. Per esempio, l'intestazione della funzione `modifyArray` si scrive così

```
void modifyArray( int b[], int arraySize )
```

a indicare che la funzione si aspetta un array di interi in corrispondenza del parametro `b`. Non è necessario specificare la dimensione dell'array tra le parentesi quadre, e anche se lo scrivete il compilatore lo ignora. Dato che gli array sono passati per riferimento, quando la funzione utilizza il parametro `b`, in realtà non fa altro che riferire l'array originario della funzione chiamante, nel nostro caso `hourlyTemperatures`. Nel Capitolo 5 introdurremo una notazione alternativa per indicare il passaggio di un array a una funzione. Come vedremo, questa notazione si basa sull'intima correlazione che esiste fra puntatori e array.

Guardate ora il prototipo di `modifyArray`. Ha un aspetto piuttosto singolare:

```
void modifyArray( int [], int );
```

Avremmo potuto scriverlo anche nella forma

```
void modifyArray( int anyArrayName[], int anyVariableName );
```

ma, come abbiamo già detto nel Capitolo 3, il compilatore ignora i nomi delle variabili nei prototipi.

Buona abitudine 4.3

Alcuni programmatori scrivono i nomi delle variabili nei prototipi di funzione per rendere i programmi più leggibili. Il compilatore, comunque, ignora questi nomi.

Ricordate che il compito di un prototipo è informare il compilatore del numero di argomenti e del tipo di ciascun argomento, nell'ordine in cui essi appaiono.

Il programma in Figura 4.14 mostra la differenza tra passaggio di un intero array e passaggio di un solo elemento. Per prima cosa visualizza i primi cinque elementi dell'array di interi `a`. Dopo passa `a` e la sua dimensione alla funzione `modifyArray`, dove ogni elemento dell'array viene moltiplicato per 2. Infine `main` visualizza nuovamente `a`. Come rileviamo dall'output, `modifyArray` ha effettivamente modificato l'array originario `a`. Dopo queste operazioni il programma visualizza il valore di `a[ 3 ]` e lo passa alla funzione `modifyElement`. Questa funzione lo moltiplica per 2 e visualizza il nuovo valore ottenuto. Osservate che nella successiva visualizzazione di `a[ 3 ]` il valore non risulta modificato, perché i singoli elementi di un array vengono passati per valore.

```
1  // Fig. 4.14: fig04_14.cpp
2  // Passaggio di array e di singoli elementi alle funzioni
3  #include <iostream.h>
4  #include <iomanip.h>
5
6  void modifyArray( int [], int );  // sembra strano
7  void modifyElement( int );
8
9  int main()
10 {
11    const int arraySize = 5;
12 int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };
13
14 cout <<"Effects of passing entire array call-by-reference:"
15      << "\n\nThe values of the original array are:\n";
16
17 for ( i = 0; i < arraySize; i++ )
18       cout << setw( 3 ) << a[ i ];
19
20       cout << endl;
21
22    // l'array a è passato per riferimento
23    modifyArray( a, arraySize );
24
25    cout << "The values of the modified array are:\n";
26
```



```
27    for ( i = 0; i < arraySize; i++ )
28       cout << setw( 3 ) << a[ i ];
29
30    cout << "\n\n\n"
31         <<"Effects of passing array element call-by-value:"
32         << "\n\nThe value of a[3] is " << a[ 3 ] << '\n';
33    // l'elemento a[3] è passato per valore
34    modifyElement( a[ 3 ] );
35
36    cout << "The value of a[3] is " << a[ 3 ] << endl;
37
38    return 0;
39 }
40
41 void modifyArray( int b[], int sizeOfArray )
42 {
43    for ( int j = 0; j < sizeOfArray; j++ )
44       b[ j ] *= 2;
45 }
46
47 void modifyElement( int e )
48 {
49    cout << "Value in modifyElement is "
50         << ( e *= 2 ) << endl;
51 }
```

```
Effects of passing entire array call-by-reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element call-by-value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Figura 4.14 Passaggio di array e di singoli elementi alle funzioni (continua)

Figura 4.14 Passaggio di array e di singoli elementi alle funzioni.

In alcune situazioni si desidera espressamente che una funzione non possa modificare liberamente gli elementi di un array. Purtroppo gli array sono passati simulando la chiamata per riferimento, per cui si tratta di una situazione difficile da controllare. Tuttavia in *C++ esiste il qualificatore* **const**, *che consente di prevenire la modifica di un argomento* all'interno di una funzione.

Se scrivete il qualificatore `const` prima di un parametro array, gli elementi dell'array devono restare costanti nel corpo della funzione, per cui qualsiasi tentativo di modificarli provoca un errore di sintassi.

Il programma in Figura 4.15 fa uso del qualificatore `const`. La funzione `tryToModifyArray` è definita con il parametro costante `const int b []`, che indica che il parametro `b` è costante e non può essere modificato. Ognuno dei tre tentativi di modificare l'array nella funzione dà quindi un errore di sintassi del compilatore. Riprenderemo il discorso su `const` nel Capitolo 7.

```
1  // Fig. 4.15: fig04_15.cpp
2  // che mostra l'uso del qualificatore di tipo const
3  #include <iostream.h>
4
5  void tryToModifyArray( const int [] );
6
7  int main()
8  {
9     int a[] = { 10, 20, 30 };
10
11    tryToModifyArray( a );
12    cout << a[0] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
13    return 0;
14 }
15
16 void tryToModifyArray( const int b[] )
17 {
18    b[ 0 ] /= 2;     // errore
19    b[ 1 ] /= 2;     // errore
20    b[ 2 ] /= 2;     // errore
21 }
```

```
Compiling FIG04_15.CPP:
Error FIG04_15.CPP 18: Cannot modify a const object
Error FIG04_15.CPP 19: Cannot modify a const object
Error FIG04_15.CPP 20: Cannot modify a const object
Warning FIG04_15.CPP 21: Parameter 'b' is never used
```

Figura 4.15 Il qualificatore di tipo const.

Errore tipico 4.10

Non dimenticate che gli array sono passati per riferimento, e che quindi possono essere modificati dalle funzioni chiamate: altrimenti potreste commettere errori logici.

Ingegneria del software 4.3

Il qualificatore `const` *si può applicare a qualsiasi parametro in una definizione di funzione per vietare a tale funzione di modificare l'argomento. Questo è un altro esempio del principio del minimo privilegio. Come regola, non consentite ad alcuna funzione di modificare un array, a meno dinon averne una reale necessità.*

4.6 Gli algoritmi di ordinamento e gli array

L'*ordinamento* dei dati è una delle applicazioni informatiche più importanti. Un banca ordina tutti i suoi conti correnti in base al loro numero per poter preparare gli estratti conto alla fine di ogni mese. Le compagnie telefoniche ordinano i loro clienti per cognome e poi per nome, in modo da poterli individuare facilmente sull'elenco. Possiamo affermare che qualsiasi organizzazione ha bisogno di ordinare i propri dati secondo un criterio, e nella maggior parte dei casi si tratta di quantità enormi di informazioni.

Gli informatici hanno compiuto grandi sforzi dal punto di vista teorico per cercare tecniche di ordinamento efficaci. In questo capitolo illustreremo uno schema di ordinamento tra i più semplici. Negli esercizi e nel Capitolo 4 del volume Tecniche Avanzate ne introdurremo altri, più complessi e veloci.

Obiettivo efficienza 4.5

Gli algoritmi più semplici sono, a volte, anche i meno efficienti. La loro virtù è la semplicità: tuttavia a volte occorre passare ad algoritmi più complessi per ottimizzare l'efficienza di un programma.

Il programma in Figura 4.16 ordina i valori dell'array di dieci elementi `a`, dal più piccolo al più grande. La tecnica utilizzata è nota come *bubble sort*, o *ordinamento a bolle*, perché i valori più piccoli (più leggeri) salgono gradualmente in cima all'array, proprio come le bolle d'aria in una bottiglia d'acqua, mentre i valori più grandi (più pesanti) scendono gradualmente in fondo all'array. La tecnica ha bisogno di passare in rassegna tutti i valori dell'array diverse volte. In ogni passaggio vengono messe a confronto tutte le coppie di elementi contigui. Se in una coppia il primo valore è minore o uguale al secondo, i valori vengono lasciati al loro posto. Se invece il primo valore è minore del secondo, i due valori sono scambiati di posto all'interno dell'array.

```
1  // Fig. 4.16: fig04_16.cpp
2  // Questo programma pone i valori di un array in ordine crescente
3
4  #include <iostream.h>
5  #include <iomanip.h>
6
7  int main()
8  {
9     const int arraySize = 10;
10    int a[ arraySize ] = { 2,6,4,8,10,12,89,68,45,37 };
11    int i, hold;
12
13    cout << "Data items in original order\n";
14
15    for ( i = 0; i < arraySize; i++ )
16       cout << setw( 4 ) << a[ i ];
17    // passaggi
18    for ( int pass = 0; pass < arraySize    1; pass++ )
19       // un passaggio
20       for ( i = 0; i < arraySize    1; i++ )
```

Figura 4.16 Ordinamento di un array con l'algoritmo di "ordinamento a bolle" (continua)

```
21
22             if ( a[ i ] > a[ i + 1 ] ) {        // un confronto
23                hold = a[ i ];                   // uno scambio
24                a[ i ] = a[ i + 1 ];
25                a[ i + 1 ] = hold;
26             }
27
28       cout << "\nData items in ascending order\n";
29
30       for ( i = 0; i < arraySize; i++ )
31          cout << setw( 4 ) << a[ i ];
32
33       cout << endl;
34       return 0;
35    }
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Figura 4.16 Ordinamento di un array con l'algoritmo di "ordinamento a bolle"

Il programma confronta prima `a[ 0 ]` con `a[ 1 ]`, quindi `a[ 1 ]` con `a[ 2 ]`, poi `a[ 2 ]` con `a[ 3 ]` e così via fino all'ultimo confronto, di `a[ 8 ]` con `a[ 9 ]`. Gli elementi sono 10, ma sono sufficienti soltanto 9 confronti.

A causa del tipo di confronto che si fa sui numeri, può succedere che in un solo passaggio un grosso numero sia spostato anche di diverse posizioni verso la fine dell'array, mentre un valore piccolo può spostarsi al massimo di una posizione verso l'inizio dell'array.

Dopo il primo passaggio possiamo essere certi che il valore più grande si trovi in fondo all'array, nella posizione `a[ 9 ]`. Al secondo passaggio avremo in `a[ 8 ]` il valore più grande fra quelli compresi fra `a[ 0 ]` e `a[ 8 ]` ma più piccolo di `a[ 9 ]`. E così via, fino al nono passaggio quando avremo in `a[ 1 ]` il valore più grande dopo tutti quelli che si trovano da `a[ 2 ]` a `a[ 9 ]`.

In questo modo avremo lasciato in `a[ 0 ]` il valore più piccolo di tutti. Osservate che sono sufficienti 9 passaggi per ordinare un array di 10 elementi.

L'ordinamento viene effettuato in un ciclo `for` nidificato. Se è necessario uno scambio di posti, lo effettuano le seguenti istruzioni

```
hold = a[ i ];
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = hold;
```

dove la variabile `hold` memorizza temporaneamente uno dei due valori da scambiare. Non è possibile effettuare uno scambio con due sole istruzioni, in questo modo

```
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = a[ i ];
```


Se, per esempio, `a[i]` vale 7 e `a[i + 1]` vale 5, dopo il primo assegnamento entrambi i valori saranno 5 e il valore 7 sarà perduto. Per questo abbiamo bisogno di una terza variabile `hold`. La virtù principale dell'ordinamento a bolle è la sua semplicità. Purtroppo la sua esecuzione è piuttosto lenta e diventa quasi insopportabile nel caso di grossi array. Negli esercizi svilupperemo delle versioni più efficienti dell'ordinamento a bolle e alcune versioni di altri algoritmi di ordinamento.

Nei corsi più avanzati studierete gli algoritmi di ordinamento e di ricerca in modo approfondito.

4.7 Il calcolo di media, mediana e moda con gli array

Il nostro prossimo esempio è leggermente più complesso. Molte società si servono dei computer per compilare e analizzare i risultati di sondaggi d'opinione e di statistiche.

Il programma in Figura 4.17 usa dell'array `response` e lo inizializza con le 99 risposte di un sondaggio: ognuna di esse è rappresentata dalla variabile a sola lettura `responseSize`. Ogni risposta è un numero compreso tra 1 e 9.

Il compito del programma è calcolare media, mediana e moda dei 99 valori.

```
1   // Fig. 4.17: fig04_17.cpp
2   // Il programma introduce l'argomento dell'analisi dei dati
3   // di un sondaggio Calcola media, mediana e moda dei dati.
4   #include <iostream.h>
5   #include <iomanip.h>
6
7   void mean( const int [], int );
8   void median( int [], int );
9   void mode( int [], int [], int );
10  void bubbleSort( int[], int );
11  void printArray( const int[], int );
12
13  int main()
14  {
15     const int responseSize = 99;
16     int frequency[ 10 ] = { 0 },
17         response[ responseSize ] =
18            { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
19              7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
20              6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
21              7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
22              6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
23              7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
24              5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
25              7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
26              7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
27              4, 5, 6, 1, 6, 5, 7, 8, 7 };
```

Figura 4.17 Programma di analisi dei risultati di un sondaggio (continua)

```
28
29     mean( response, responseSize );
30     median( response, responseSize );
31     mode( frequency, response, responseSize );
32
33     return 0;
34  }
35
36  void mean( const int answer[], int arraySize )
37  {
38     int total = 0;
39
40     cout << "********\n  Mean\n********\n";
41
42     for ( int j = 0; j < arraySize; j++ )
43        total += answer[ j ];
44
45     cout << "The mean is the average value of the data\n"
46          << "items. The mean is equal to the total of\n"
47          << "all the data items divided by the number\n"
48          << "of data items (" << arraySize
49          << "). The mean value for\nthis run is: "
50          << total << " / " << arraySize << " = "
51          << setiosflags( ios::fixed | ios::showpoint )
52          << setprecision( 4 ) << ( float ) total / arraySize
53          << "\n\n";
54  }
55
56  void median( int answer[], int size )
57  {
58     cout << "\n********\n Median\n********\n"
59          << "The unsorted array of responses is";
60
61     printArray( answer, size );
62     bubbleSort( answer, size );
63     cout << "\n\nThe sorted array is";
64     printArray( answer, size );
65     cout << "\n\nThe median is element " << size / 2
66          << " of\nthe sorted " << size
67          << " element array.\nFor this run the median is "
68          << answer[ size / 2 ] << "\n\n";
69  }
70
71  void mode( int freq[], int answer[], int size )
72  {
73     int rating, largest = 0, modeValue = 0;
74
75     cout << "\n********\n  Mode\n********\n";
```

Figura 4.17 *Programma di analisi dei risultati di un sondaggio (continua)*


```
76
77     for ( rating = 1; rating <= 9; rating++ )
78        freq[ rating ] = 0;
79
80     for ( int j = 0; j < size; j++ )
81        ++freq[ answer[ j ] ];
82
83     cout << "Response"<< setw( 11 ) << "Frequency"
84          << setw( 19 ) << "Histogram\n\n" << setw( 55 )
85          << "1    1    2    2\n" << setw( 56 )
86          << "5    0    5    0    5\n\n";
87
88     for ( rating = 1; rating <= 9; rating++ ) {
89        cout << setw( 8 ) << rating << setw( 11 )
90             << freq[ rating ] << "          ";
91
92        if ( freq[ rating ] > largest ) {
93           largest = freq[ rating ];
94           modeValue = rating;
95        }
96
97        for ( int h = 1; h <= freq[ rating ]; h++ )
98           cout << '*';
99
100       cout << '\n';
101    }
102
103    cout << "The mode is the most frequent value.\n"
104         << "For this run the mode is " << modeValue
105         << "which occurred " << largest << " times."
106         << endl;
107 }
108
109
110 void bubbleSort( int a[], int size )
111 {
112    int hold;
113
114    for ( int pass = 1; pass < size; pass++ )
115
116       for ( int j = 0; j < size - 1; j++ )
117
118          if ( a[ j ] > a[ j + 1 ] ) {
119             hold = a[ j ];
120             a[ j ] = a[ j + 1 ];
121             a[ j + 1 ] = hold;
122          }
123 }
```

Figura 4.17 *Programma di analisi dei risultati di un sondaggio (continua)*

```
124
125  void printArray( const int a[], int size )
126  {
127     for ( int j = 0; j < size; j++ ) {
128
129        if ( j % 20 == 0 )
130           cout << endl;
131
132        cout << setw( 2 ) << a[ j ];
133     }
134  }
```

Figura 4.17 Programma di analisi dei risultati di un sondaggio.

Per la media aritmetica di 99 valori, calcolata dalla funzione `mean`, occorre sommare i 99 valori e dividere il risultato per 99.

La mediana è anche detta valore centrale della serie. La funzione `median` *la determina* chiamando prima la funzione `bubbleSort`, per ordinare l'array dal valore più piccolo al più grande, e successivamente scegliendo il valore centrale dell'array ordinato, cioè `answer[responseSize / 2]`.

Nel caso di un numero pari di elementi la mediana è definita come la media dei due valori centrali, ma la nostra funzione `median` non prevede questa eventualità. L'array `response` viene poi visualizzato dalla funzione `printArray`.

La moda è il valore che si ripete più volte nella serie delle 99 risposte. La funzione `mode` conta il numero di risposte di ciascun tipo e seleziona il valore che ha conseguito il conteggio più alto.

Questa versione di `mode` *non considera l'eventualità di singolarità (cfr. l'esercizio 4.14).* La funzione `mode` produce anche un istogramma, per consentire di determinare la moda anche graficamente.

In Figura 4.18 abbiamo un'esecuzione del programma. In questo programma abbiamo incluso la maggior parte delle operazioni in gioco nel trattamento di array, compreso il *passaggio di array a funzioni.*

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788
```

Figura 4.18 Output del programma di analisi dei risultati di un sondaggio (continua)


```
********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7
The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
  Mode
********
Response  Frequency              Histogram
                                       1    1    2    2
                                  5    0    5    0    5

       1          1              *
       2          3              ***
       3          4              ****
       4          5              *****
       5          8              ********
       6          9              *********
       7         23              ***********************
       8         27              ***************************
       9         19              *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Figura 4.18 Output del programma di analisi dei risultati di un sondaggio.

4.8 Le ricerche in array: ricerca lineare e binaria

Spesso ci si trova a lavorare su grandi quantità di dati memorizzati in grossi array. A volte si ha bisogno di determinare se un array contiene un valore uguale a un certo *valore chiave*. Il procedimento con cui si verifica la presenza di un valore in un array viene detto *ricerca*. In questa sezione discuteremo due tecniche di ricerca, una semplice, la *ricerca lineare*, e una più efficiente, la *ricerca binaria*. Negli esercizi 4.33 e 4.34 vi sarà chiesto di implementare le versioni ricorsive di questi due algoritmi di ricerca.

Nella ricerca lineare (Figura 4.19) si procede confrontando ciascun elemento dell'array con il *valore chiave*. Se gli elementi dell'array non si trovano in un ordine particolare, il valore cercato può essere il primo ma anche l'ultimo. In media, quindi, è necessario un numero di confronti pari alla metà del numero di elementi dell'array.

```
1  // Fig. 4.19: fig04_19.cpp   Ricerca lineare in un array
2  #include <iostream.h>
3
4  int linearSearch( const int [], int, int );
5
6  int main()
7  {
8     const int arraySize = 100;
9     int a[ arraySize ], searchKey, element;
10
11    for (int x = 0; x < arraySize; x++)  // crea alcuni dati
12       a[ x ] = 2 * x;
13
14    cout << "Enter integer search key:" << endl;
15    cin >> searchKey;
16    element = linearSearch( a, searchKey, arraySize );
17
18    if ( element != -1 )
19       cout << "Found value in element " << element << endl;
20    else
21       cout << "Value not found" << endl;
22
23    return 0;
24 }
25
26 int linearSearch( const int array[], int key,
27                   int sizeOfArray )
28 {
29    for ( int n = 0; n < sizeOfArray; n++ )
30       if ( array[ n ] == key )
31          return n;
32
33    return -1;
34 }
```

```
Enter integer search key:
36
Found value in element 18

Enter integer search key:
37
Value not found
```

Figura 4.19 Ricerca lineare su un array.

Per verificare che il valore non sia nell'array, l'algoritmo deve purtroppo effettuare il confronto su tutti gli elementi dell'array. La ricerca lineare è abbastanza utile nel caso di array piccoli o non ordinati. Tuttavia è inadatta ad array di grandi dimensioni. Se l'array è già ordinato possiamo utilizzare la tecnica più efficiente di ricerca binaria.

L'algoritmo di ricerca binaria scarta metà degli elementi dell'array dopo ogni confronto. Esso localizza l'elemento centrale e lo confronta con il valore chiave. Se i due valori sono uguali, vuol dire che la chiave di ricerca è stata trovata, e la funzione restituisce l'indice dell'elemento confrontato. Altrimenti, il problema si riduce a effettuare la ricerca in metà dell'array. Se la chiave di ricerca ha un valore minore dell'elemento centrale dell'array, la ricerca continua nella prima metà dell'array, e la seconda metà viene scartata, perché conterrà valori che sono tutti maggiori della chiave. L'algoritmo ora localizza il valore centrale della prima metà dell'array e lo confronta con la chiave di ricerca. Se i due valori sono uguali la ricerca si intende terminata, altrimenti viene eliminato un quarto dei valori dell'array e la ricerca continua nell'altro quarto, e così via.

Per avere un'idea dell'efficienza di questo algoritmo consideriamo che il caso peggiore per un array di 1024 elementi prevede soltanto 10 confronti. Infatti dividendo ripetutamente 1024 per 2 (a ogni confronto scartiamo metà dell'array) otteniamo i valori 512, 256, 128, 64, 32, 16, 8, 4, 2 e 1. Il numero 1024, decima potenza di 2, può essere diviso per 2 soltanto 10 volte prima di arrivare al valore 1. La divisione per 2 corrisponde a un confronto nell'algoritmo di ricerca binaria. Per un array di 1048576 elementi, ventesima potenza di 2, sono necessari al massimo 20 confronti per trovare il valore chiave. Per un miliardo di elementi abbiamo bisogno soltanto di 30 confronti. Come vedete l'incremento dell'efficienza è esponenziale rispetto all'algoritmo di ricerca lineare, dove in media dovevamo effettuare i confronti su metà degli elementi. Per un miliardo di elementi, la differenza è tra 500 milioni di confronti e 30 (!). Il numero massimo di confronti necessari per una ricerca binaria si trova determinando la prima potenza di 2 maggiore del numero degli elementi dell'array.

Obiettivo efficienza 4.6

L'efficienza della ricerca binaria, incredibilmente maggiore di quella della ricerca lineare, non ha costo zero. L'ordinamento di un array, infatti, è molto più dispendioso rispetto alla ricerca di un solo valore chiave. In generale conviene ordinare l'array solo se si prevede di effettuare numerose ricerche su di esso.

La Figura 4.20 presenta una versione iterativa della funzione `binarySearch`. Gli argomenti richiesti sono quattro, un array di interi `b`, un intero `searchKey` (chiave di ricerca) e gli indici `low` (indice iniziale) e `high` (indice finale), che sono il primo indice dal quale iniziare la ricerca e l'ultimo indice, dove fermarsi. Se la chiave di ricerca non è uguale all'elemento centrale di un sottoarray, gli indici `low` e `high` vengono regolati di conseguenza, in modo che la ricerca prosegua sul successivo sottoarray più piccolo. Se la chiave di ricerca è minore dell'elemento centrale, l'indice `high` assume il valore di `middle - 1`, l'indice del valore centrale meno uno, e la ricerca continua sugli elementi compresi tra gli indici `low` e `middle - 1`. Se la chiave di ricerca è maggiore dell'elemento centrale, invece, l'indice `low` prende il valore di `middle + 1` e la ricerca continua sugli elementi compresi tra gli indici `middle + 1` e `high`. L'array del programma contiene 15 elementi; la prima potenza di 2 maggiore di questo numero è 16, quarta potenza di 2, quindi avremo un massimo di 4 confronti prima di trovare la chiave di ricerca. La funzione `printHeader`

visualizza gli indici dell'array mentre la funzione printRow visualizza ciascun sottoarray creato dalla ricerca binaria. L'elemento centrale di ciascun sottoarray viene marcato con un asterisco, per segnalare l'elemento che viene confrontato con la chiave.

```
1   // Fig. 4.20: fig04_20.cpp
2   // Ricerca binaria in un array
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int binarySearch( int [], int, int, int, int );
7   void printHeader( int );
8   void printRow( int [], int, int, int, int );
9
10  int main()
11  {
12     const int arraySize = 15;
13     int a[ arraySize ], key, result;
14
15     for ( int i = 0; i < arraySize; i++ )
16        a[ i ] = 2 * i;    // pone alcuni dati nell'array
17
18     cout << "Enter a number between 0 and 28: ";
19     cin >> key;
20
21     printHeader( arraySize );
22     result = binarySearch(a,key,0,arraySize    1,arraySize);
23
24     if ( result != -1 )
25        cout << '\n' << key << " found in array element "
26             << result << endl;
27     else
28        cout << '\n' << key << " not found" << endl;
29
30     return 0;
31  }
32  // Ricerca binaria
33  int binarySearch(int b[],int searchKey,int low,int high,int size)
34  {
35     int middle;
36
37     while ( low <= high ) {
38        middle = ( low + high ) / 2;
39
40        printRow( b, low, middle, high, size );
41  // confronto con l'elemento centrale dell'array
42        if ( searchKey == b[ middle ] )
43           return middle;
```

Figura 4.20 *Ricerca binaria su un array ordinato (continua)*


```
44        else if ( searchKey < b[ middle ] )
45           high = middle    1;      // ricerca sulla metà inferiore
46        else
47           low = middle + 1;       // ricerca sulla metà superiore
48     }
49
50     return -1;          // la chiave di ricerca non e stata trovata
51  }
52
53  // Visualizza un'intestazione per l'output
54  void printHeader( int size )
55  {
56     cout << "\nSubscripts:\n";
57     for ( int i = 0; i < size; i++ )
58        cout << setw( 3 ) << i << ' ';
59
60     cout << '\n';
61
62     for ( i = 1; i <= 4 * size; i++ )
63        cout << '-';
64
65     cout << endl;
66  }
67  // Visualizza una riga di output che mostra la parte
68  // dell'array che e in elaborazione in un dato momento.
69  void printRow( int b[],int low,int mid,int high,int size )
70  {
71     for ( int i = 0; i < size; i++ )
72        if ( i < low || i > high )
73           cout << "    ";
74        else if ( i == mid )       // marca il valore centrale
75           cout << setw( 3 ) << b[ i ] << '*';
76        else
77           cout << setw( 3 ) << b[ i ] << ' ';
78
79     cout << endl;
80  }
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
---------------------------------------------
  0  2  4  6  8 10 12 14* 16 18 20 22 24 26 28
                    16 18 20 22* 24 26 28
                                24 25* 28
                                   24*

25 not found
```

Figura 4.20 Ricerca binaria su un array ordinato (continua)

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*
8 found in array element 4


Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

Figura 4.20 Ricerca binaria su un array ordinato.

4.9 Gli array multidimensionali

In C++ un array può avere più di un indice, e quindi più di una dimensione. Un uso comune degli array multidimensionali è la rappresentazione di tabelle di valori (o *matrici*), con i dati disposti su righe e colonne. Per identificare un determinato elemento di una matrice servono due indici: il primo, per convenzione, identifica la riga dell'elemento desiderato, mentre il secondo ne identifica la colonna..

Una tabella dunque può essere rappresentata con un array a 2 indici. Tuttavia gli array possono avere anche più di due indici. I compilatori supportano generalmente almeno 12 indici. La Figura 4.21 illustra l'array a due indici `a`. L'array è composto di tre righe e quattro colonne, per cui diciamo brevemente che si tratta di un array *3x4*. In generale, un array con *m* righe e *n* colonne viene detto array *mxn*.

Ogni elemento dell'array `a` è identificato in Figura 4.21 dal nome dell'elemento nella forma `a[ x ][ y ]`, dove `a` è il nome dell'array mentre `x` e `y` sono gli indici che identificano univocamente l'elemento in `a`. Osservate che gli elementi della prima riga hanno tutti il primo indice uguale a `0`, mentre quelli della quarta riga hanno tutti il primo indice uguale a `3`.

Errore tipico 4.11

Fate attenzione a non riferirsi ad un elemento di una matrice in modo scorretto, per esempio scrivendo `a[ x ][ y ]` nella forma `a[x, y]`. Infatti `a[x, y]` avrebbe lo stesso significato di `a[y]` perché in C++ la virgola è un operatore che dà come risultato il suo secondo operando (in questo caso `y`).

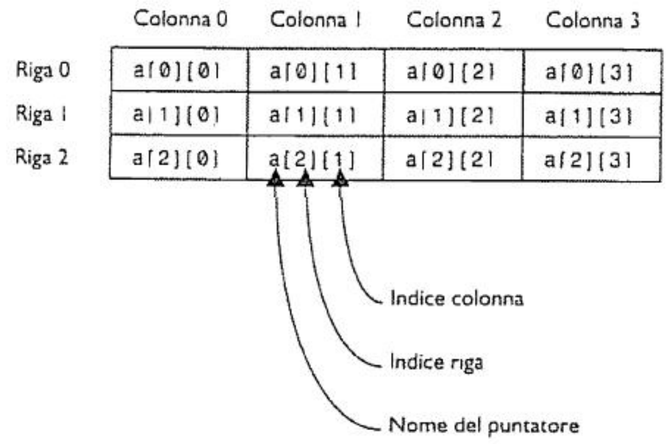


Figura 4.21 Un array bidimensionale con tre righe e quattro colonne.

Un array multidimensionale può essere inizializzato in fase di dichiarazione, proprio come un array unidimensionale. Per esempio, l'array a due indici `b[ 2 ][ 2 ]` può essere dichiarato e inizializzato così:

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

I valori sono raggruppati per righe tra parentesi graffe. I valori `1` e `2` inizializzano l'elemento `b[ 0 ][ 0 ]` e `b[ 0 ][ 1 ]` mentre i valori `3` e `4` inizializzano `b[ 1 ][ 0 ]` e `b[ 1 ][ 1 ]`. Se in corrispondenza di una riga non sono presenti tutti gli inizializzatori, quelli mancanti sono azzerati per default. La dichiarazione

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

inizializza `b[ 0 ][ 0 ]` a 1, `b[ 0 ][ 1 ]` a `0`, `b[ 1 ][ 0 ]` a 3 e `b[ 1 ][ 1 ]` a 4.

La Figura 4.22 mostra la dichiarazione e l'inizializzazione di array a due indici. Il programma dichiara tre array, ognuno dei quali ha due righe e tre colonne. Nella dichiarazione di `array1` sono presenti sei inizializzatori in due sottoliste. La prima sottolista inizializza la prima riga dell'array con i valori `1`, `2` e `3` mentre la seconda sottolista inizializza la seconda riga con i valori `4`, `5` e `6`. Se togliamo le parentesi graffe che racchiudono le due sottoliste il compilatore inizializza automaticamente gli elementi della prima riga e poi gli elementi della seconda.

Nella dichiarazione di `array2` sono presenti cinque inizializzatori. Gli elementi per cui non è stata prevista un'inizializzazione esplicita sono azzerati, per cui `array2[ 1 ][ 2 ]` vale `0`.

```
1  // Fig. 4.22: fig04_22.cpp
2  // Inizializzazione di array multidimensionali
3  #include <iostream.h>
4
5  void printArray( int [][ 3 ] );
6
7  int main()
```

Figura 4.22 Inizializzazione di array multidimensionali (continua)

```
 8  {
 9     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } },
10         array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 },
11         array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
12
13     cout << "Values in array1 by row are:" << endl;
14     printArray( array1 );
15
16     cout << "Values in array2 by row are:" << endl;
17     printArray( array2 );
18
19     cout << "Values in array3 by row are:" << endl;
20     printArray( array3 );
21
22     return 0;
23  }
24
25  void printArray( int a[][ 3 ] )
26  {
27     for ( int i = 0; i < 2; i++ ) {
28
29        for ( int j = 0; j < 3; j++ )
30           cout << a[ i ][ j ] << ' ';
31
32        cout << endl;
33     }
34  }
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Figura 4.22 Inizializzazione di array multidimensionali.

Nella dichiarazione di `array3` sono presenti tre inizializzatori in due sottoliste. La sottolista per la prima riga inizializza esplicitamente i primi due elementi della prima riga a 1 e 2. Il terzo elemento viene azzerato automaticamente. La sottolista per la seconda riga inizializza esplicitamente il primo elemento a 4. Gli altri due elementi vengono azzerati.

Il programma chiama la funzione `printArray` per visualizzare gli elementi di ciascun array. Notate che nella definizione della funzione l'array viene indicato con `int a[][ 3 ]`. Se una funzione riceve un array unidimensionale, le parentesi quadre possono essere lasciate vuote nella lista di parametri. Allo stesso modo la prima dimensione di un array non è necessaria, ma fate attenzione, perché tutte le successive invece lo sono. Il compilatore

utilizza le dimensioni successive alla prima per determinare le locazioni di memoria dove iniziano e finiscono, per esempio, le righe e le colonne, nel caso degli array a due dimensioni. Ricordate inoltre che gli elementi di un array sono posti in locazioni di memoria consecutive, indipendentemente dal numero di dimensioni dell'array. Nel caso di un array bidimensionale, in memoria troviamo prima la prima riga e poi la seconda.

Per localizzare un elemento di una determinata riga la funzione deve sapere di quanti elementi è composta ogni riga, in modo tale da saltare il giusto numero di locazioni di memoria. Se deve accedere, per esempio, a `a[ 1 ][ 2 ]` la funzione sa che deve saltare tre elementi in memoria per poter accedere alla seconda riga (riga di indice 1). Dopo di che può accedere al terzo elemento di quella riga (elemento di indice 2).

Nelle manipolazioni di array sono molto utilizzati i costrutti `for`. Per esempio, il seguente costrutto `for` azzera tutti gli elementi della terza riga di `a` (cfr. Figura 4.21):

```
for ( column = 0; column < 4; column++ )
   a[ 2 ][ column ] = 0;
```

Abbiamo detto la *terza* riga, per cui l'indice corrispondente sarà 2, dato che gli indici partono da 0. Il ciclo `for` fa variare solo il secondo indice, cioè l'indice di colonna. Il ciclo effettua un'operazione equivalente a questa serie di assegnamenti:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

Il prossimo ciclo `for` nidificato calcola la somma degli elementi di `a` in `total`:

```
total = 0;
for ( row = 0; row < 3; row++ )
   for ( column = 0; column < 4; column++ )
      total += a[ row ][ column ];
```

Questo codice somma uno per volta gli elementi dell'array. Il costrutto `for` esterno azzera innanzitutto `row`, l'indice di riga, così il costrutto `for` interno può sommare tutti gli elementi della prima riga. Successivamente il costrutto `for` esterno incrementa `row` a 1, per consentire la somma degli elementi della seconda riga, e così via.

Il programma in Figura 4.23 effettua alcune manipolazioni comuni sull'array 3x4 `studentGrade`. Ogni riga dell'array rappresenta uno studente e ogni colonna rappresenta i voti conseguiti dallo studente nei quattro esami di un dato semestre. Abbiamo previsto quattro funzioni per le manipolazioni sull'array. La funzione `minimum` determina il voto più basso conseguito da ogni studente, mentre `maximum` ne determina il voto più alto. La funzione `average` determina la media dei voti di un determinato studente. Per ultima troviamo la funzione `printArray`, che visualizza tutto l'array in formato tabulare.

```
1  // Fig. 4.23: fig04_23.cpp
2  // Esempio di array bidimensionale
3  #include <iostream.h>
4  #include <iomanip.h>
5
6  const int students = 3;          // numero di studenti
```

Figura 4.23 Esempio di utilizzo di array bidimensionali (continua)

```
const int exams = 4;             // numero di esami

int minimum( int [][ exams ], int, int );
int maximum(int [][ exams ], int, int );
float average( int [], int );
void printArray( int [][ exams ], int, int );

int main()
{
   int studentGrades[ students ][ exams ] =
          { { 77, 68, 86, 73 },
            { 96, 87, 89, 78 },
            { 70, 90, 86, 81 } };

   cout << "The array is:\n";
   printArray( studentGrades, students, exams );
   cout << "\n\nLowest grade: "
        << minimum( studentGrades, students, exams )
        << "\nHighest grade: "
        << maximum(studentGrades, students, exams) << '\n';

   for ( int person = 0; person < students; person++ )
      cout << " The average grade for student "
           << person << " is "
           << setiosflags( ios::fixed | ios::showpoint )
           << setprecision( 2 )
           << average( studentGrades[ person ], exams )
           << endl;

   return 0;
}

// Trova il voto minimo
int minimum( int grades[][ exams ], int pupils, int tests )
{
   int lowGrade = 100;

   for ( int i = 0; i < pupils; i++ )

      for ( int j = 0; j < tests; j++ )

         if ( grades[ i ][ j ] < lowGrade )
            lowGrade = grades[ i ][ j ];

   return lowGrade;
}

// Trova il voto massimo
int maximum( int grades[][ exams ], int pupils, int tests )
```



```
{
   int highGrade = 0;

   for ( int i = 0; i < pupils; i++ )

      for ( int j = 0; j < tests; j++ )

         if ( grades[ i ][ j ] > highGrade )
            highGrade = grades[ i ][ j ];

   return highGrade;
}

// Determina il voto medio per uno studente particolare
float average( int setOfGrades[], int tests )
{
   int total = 0;

   for ( int i = 0; i < tests; i++ )
      total += setOfGrades[ i ];

   return ( float ) total / tests;
}

// Visualizza l'array
void printArray(int grades[][exams], int pupils, int tests)
{
   cout << "                  [0]  [1]  [2]  [3]";

   for ( int i = 0; i < pupils; i++ ) {
      cout << "\nstudentGrades[" << i << "] ";

      for ( int j = 0; j < tests; j++ )
         cout << setiosflags( ios::left ) << setw( 5 )
              << grades[ i ][ j ];
   }
}
```

```
The array is:
                  [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81
Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Figura 4.23 Esempio di utilizzo di array bidimensionali (continua)

Figura 4.23 Esempio di utilizzo di array bidimensionali.

Le funzioni `minimum`, `maximum` e `printArray` ricevono ognuna tre argomenti, l'array `studentGrade` (chiamato `grades` nella lista degli argomenti), il numero di studenti (ovvero di righe dell'array) e il numero di esami (ovvero di colonne dell'array). Ciascuna funzione contiene un ciclo `for` che manipola l'array ricevuto. Questo è il costrutto `for` della funzione `minimum`:

```
for ( i = 0; i < pupils; i++ )
   for ( j = 0; j < tests; j++ )
      if ( grades[ i ][ j ] < lowGrade )
         lowGrade = grades[ i ][ j ];
```

Il `for` esterno azzera innanzitutto la variabile `i`, cioè l'indice di riga, così gli elementi della prima riga possono essere confrontati alla variabile `lowGrade` nel corpo del `for` interno, per determinare il valore minimo. Alla fine del ciclo nidificato `lowGrade` contiene il voto più basso dell'array. La funzione `maximum` funziona in modo analogo.

La funzione `average` prende due argomenti, un array unidimensionale dei risultati degli esami relativi a un particolare studente e il numero di risultati presenti nell'array. Quando viene chiamata `average` il primo argomento è `studentGrade[ student ]`, cioè una riga specifica dell'array bidimensionale. Per esempio l'argomento `studentGrades[ 1 ]` rappresenta l'array unidimensionale dei quattro valori che si trovano sulla seconda riga di `studentGrade`. Un array bidimensionale può essere visto come un array i cui elementi sono array unidimensionali. La funzione `average` somma gli elementi dell'array unidimensionale, divide il totale per il numero di esami e restituisce un risultato a virgola mobile.

4.10 Pensare in termini di oggetti: come identificare le operazioni di una classe [progetto opzionale]

Nelle sezioni "Pensare in termini di oggetti" dei Capitoli 2 e 3 abbiamo introdotto i primi passi della progettazione orientata agli oggetti. Nel Capitolo 2 abbiamo identificato le classi da implementare e abbiamo rappresentato la struttura del nostro sistema ponendole in un diagramma delle classi. Nel Capitolo 3 abbiamo determinato diversi attributi di tali classi e abbiamo studiato i possibili stati della classe `Elevator` rappresentandoli in un diagramma di stato. Infine abbiamo rappresentato in un diagramma delle attività la logica seguita dall'ascensore in risposta alla pressione dei pulsanti.

In questa sezione cercheremo di determinare le *operazioni* (o comportamenti) del nostro simulatore mentre nel Capitolo 5 ci concentreremo sulle collaborazioni (interazioni) tra gli oggetti delle classi.

Un'operazione è un servizio che una classe fornisce ai propri utenti (detti anche clienti). Per capire meglio il concetto, torniamo alle analogie con il mondo reale. Ad esempio, tra le operazioni di una radio troviamo l'impostazione della stazione e del volume, che sono normalmente invocate da un ascoltatore che opera sulle manopole dell'apparecchio. Tra quelle di un'automobile possiamo considerare l'accelerazione, ottenuta in seguito alla pressione del pedale dell'acceleratore o la decelerazione, ottenuta con la pressione del pedale del freno, la sterzata e il cambio delle marce.

Gli oggetti normalmente non effettuano le operazioni in modo spontaneo: accade invece che una specifica operazione venga invocata quando un oggetto "origine" (detto anche *oggetto client*) invia un *messaggio* a un oggetto "destinazione" (detto anche *oggetto server*), chiedendo a quest'ultimo di effettuare l'operazione in questione. Ciò vi ricorderà la chiamata di una funzione membro, che più precisamente è il modo in cui il C++ implementa l'invio dei messaggi agli oggetti. In questa sezione cercheremo di individuare la maggior parte delle operazioni che le nostre classi devono poter offrire ai propri clienti.

Partendo semplicemente dalla definizione del problema, possiamo individuare diverse operazioni per ogni classe. Per fare ciò prendiamo in esame i verbi e le espressioni operative, e cerchiamo di correlare ognuna di queste espressioni a una classe specifica del nostro sistema (cfr. Fig. 4.24). Nella tabella in Fig. 4.24 abbiamo trascritto alcune espressioni.

Classe	Verbi ed espressioni operative
`Elevator`	si sposta, arriva al piano, rilascia il pulsante esterno, suona il campanello, segnala il suo arrivo al piano, apre la porta, chiude la porta
`Clock`	emette un tic ogni istante
`Scheduler`	programma due istanti temporali a caso, crea una persona, dice alla persona di accedere al piano, verifica che il piano sia libero, ritarda di un istante la creazione di una persona
`Person`	scende al piano, preme il pulsante esterno, preme il pulsante interno, entra nell'ascensore, esce dall'ascensore
`Floor`	rilascia il pulsante esterno, spegne la spia, accende la spia
`FloorButton`	chiama l'ascensore
`ElevatorButton`	segnala all'ascensore di spostarsi
`Door`	(apertura) segnala alla persona di uscire dall'ascensore, (apertura) *segnala alla persona di entrare nell'ascensore*
`Bell`	-
`Light`	-
`Building`	-

Figura 4.24 Frasi operative per ogni classe del simulatore.

Per ottenere le operazioni, dobbiamo esaminare le espressioni annotate in corrispondenza di ogni classe. Ad esempio, il verbo "si sposta" della classe `Elevator` si riferisce all'attività durante la quale l'ascensore effettua una corsa da un piano all'altro. Dobbiamo rendere "si sposta" un'operazione della classe `Elevator`? Notiamo che non c'è alcun messaggio che dica all'ascensore di spostarsi, è piuttosto l'ascensore che decide di farlo autonomamente, in risposta alla pressione del pulsante e a condizione che la porta sia chiusa. Per questo motivo "*si sposta*" *non corrisponde a un'operazione*. Neanche l'espressione "arriva al piano" è un'operazione, perché l'ascensore decide da sé quando è arrivato al piano, sulla base del valore del timer. L'espressione "rilascia il pulsante interno" implica invece che l'ascensore invii un messaggio al pulsante interno, ordinandogli di rilasciarsi: perciò la classe `ElevatorButton` ha bisogno di un'operazione per fornire questo servizio all'ascen-

sore. Nel diagramma delle classi poniamo questa operazione nella parte inferiore della classe `ElevatorButton` (Figura 4.25). I nomi delle operazioni diventano funzioni, comprensive di nome e tipo restituito:

```
resetButton() : void
```

La notazione prevede prima il nome dell'operazione, seguito da una coppia di parentesi tonde che contiene un elenco di parametri separati da virgole necessari all'operazione (nel nostro caso, nessuno). La lista dei parametri è seguita da un segno di due punti e dal tipo restituito dall'operazione (nel nostro caso `void`). Notate che la maggior parte delle nostre operazioni non sembra avere parametri e restituire un tipo diverso da `void`, tuttavia le cose potrebbero cambiare nelle fasi successive della nostra progettazione.

Dall'espressione "suona il campanello" relativa alla classe `Elevator` concludiamo che la classe `Bell` deve avere un'operazione che fornisce questo servizio, e cioè per l'appunto il suono: annotiamo perciò l'operazione `ringBell` (suona il campanello) sotto la classe `Bell`.

Quando l'ascensore arriva al piano esso "segnala il suo arrivo al piano", e il piano risponde effettuando diverse attività, tra cui il rilascio del pulsante esterno e lo spegnimento della spia.

Perciò la classe `Floor` ha bisogno di avere un'operazione che fornisca questo servizio: la chiameremo `elevatorArrived` (ascensore arrivato) e ne scriveremo il nome nella parte inferiore della classe `Floor` in Figura 4.25.

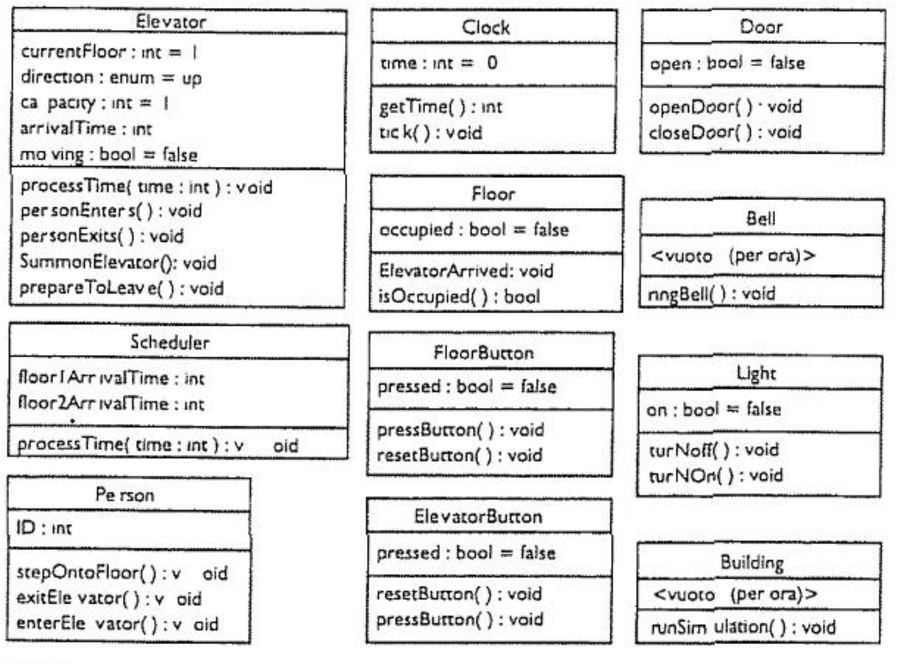


Figura 4.25 Diagramma delle classi con attributi e operazioni.

Le due espressioni restanti che abbiamo annotato in corrispondenza della classe `Elevator` dicono che l'ascensore ha bisogno di aprire e chiudere la porta. È la classe `Door` che deve fornire queste operazioni, che chiameremo `openDoor` (apri la porta) e `closeDoor` (chiudi la porta) e porremo nella parte inferiore della classe `Door`.

In corrispondenza della classe `Clock` leggiamo la frase "emette un tic ad istanti regolari" Questa frase pone un interrogativo interessante: è naturale che l'operazione "ottenere l'orario" sia fornita dalla classe del timer, ma l'emissione del tic, per così dire, è essa stessa un'operazione? Per rispondere consideriamo il modo in cui si svolge la nostra simulazione.

La definizione del problema indica che il meccanismo di gestione degli arrivi debba conoscere il valore temporale corrente per sapere se è ora di creare una nuova persona su uno dei piani. L'ascensore ha bisogno della stessa informazione per decidere se è arrivato o meno alla fine della sua corsa. Infine abbiamo deciso che l'edificio avrà la responsabilità di eseguire la simulazione e di passare il valore temporale corrente al meccanismo di gestione degli arrivi e all'ascensore. A questo punto dovremmo avere un'idea di come si svolge la simulazione: l'edificio ripete i seguenti passi ad ogni istante misurato dal timer per tutta la durata della simulazione:

1. Legge l'istante corrente dal timer.
2. Passa il valore temporale corrente al meccanismo di gestione degli arrivi, in modo che quest'ultimo possa creare una nuova persona, se necessario.
3. Passa il valore temporale corrente all'ascensore, in modo che quest'ultimo possa sapere se la sua corsa è terminata, nel caso si stia spostando.

Abbiamo deciso che l'edificio avrà dunque la completa responsabilità di gestire le varie parti della simulazione. Ciò significa che dovrà essere l'edificio a incrementare il timer ad ogni istante, e al termine di questa operazione il valore temporale corrente dovrà essere passato al meccanismo di gestione degli arrivi e all'ascensore. Ciò ci suggerisce di creare due operazioni, `getTime` (ottieni l'istante corrente) e `tick` (passa all'istante successivo), e di associarle alla classe `Clock`. L'operazione `getTime` restituisce un `int` che rappresenta l'attributo "istante corrente" del timer. Nei passi 2 e 3 leggiamo "Passa il valore temporale al programmatore" e "Passa il valore temporale all'ascensore" possiamo dunque aggiungere l'operazione `processTime` (elabora il valore temporale) alle classi `Scheduler` e `Elevator`. Possiamo aggiungere infine l'operazione `runSimulation` (effettua la simulazione) alla classe `Building`.

La classe `Scheduler` presenta le espressioni "genera due istanti temporali a caso" e "ritarda di un istante la creazione di una persona". Il programmatore decide autonomamente di effettuare queste azioni e non fornisce tali servizi ai clienti, e dunque queste espressioni non descrivono delle operazioni.

L'espressione "crea una persona" della classe `Scheduler` presenta un caso speciale. Anche se possiamo rappresentare un oggetto della classe `Scheduler` che invia un messaggio "crea", un oggetto della classe `Person` non può rispondere a tale messaggio in quanto l'oggetto in questione non esiste ancora. La creazione degli oggetti riguarda i dettagli di implementazione e non viene rappresentata come operazione di una classe. Ne riparleremo nel Capitolo 7, quando studieremo la creazione di nuovi oggetti.

L'espressione "dice alla persona di accedere al piano" in Figura 4.24 significa che la classe `Person` deve avere un'operazione che il programmatore può invocare per dire alla *persona di accedere al piano*. *Chiameremo questa operazione* `stepOntoFloor` (accedi al piano) e la associamo alla classe `Person`.

L'espressione "verifica che il piano sia libero" implica che la classe `Floor` deve fornire un servizio che consente agli oggetti del sistema di sapere se il piano è occupato o meno. L'operazione che creeremo per questo servizio restituirà `true` se il piano è occupato, e `false` altrimenti. Inseriamo dunque

```
isOccupied( ) : bool
```

fra le operazioni della classe `Floor`.

La classe `Person` elenca le espressioni "preme il pulsante esterno" e "preme il pulsante interno". L'operazione `pressButton` (premi pulsante) andrà quindi nelle classi `FloorButton` ed `ElevatorButton` nel nostro diagramma delle classi UML (Figura 4.25). Abbiamo già incontrato l'evento "persona che accede al piano" quando abbiamo analizzato le espressioni *relative alla classe* `Scheduler`, *per cui non dobbiamo creare ulteriori operazioni per lo* stesso evento elencato nella classe `Person`. Le espressioni "entra nell'ascensore" ed "esce dall'ascensore" relative alla classe `Person` ci dicono che la classe `Elevator` necessita delle operazioni corrispondenti [Nota: a questo punto possiamo soltanto immaginare cosa fanno queste operazioni. Per esempio, probabilmente rappresentano ascensori reali con un sensore che rileva l'entrata e l'uscita del passeggero. Per ora ci limitiamo ad elencarle, scopriremo più in là se queste operazioni compiono delle azioni (e quali sono) quando implementeremo il simulatore in C++.]

Nelle espressioni della classe `Floor` troviamo anche "rilascia il pulsante esterno", perciò scriviamo l'operazione relativa `resetButton` (rilascia pulsante) nella classe `FloorButton`. Per la classe `Floor` troviamo anche "spegne la spia" e "accende la spia", per cui creiamo le operazioni `turnOff` (spegni) e `turnOn` (accendi) e le comprendiamo fra quelle della classe `Light`.

L'espressione "chiama l'ascensore" relativa a `FloorButton` implica che la classe `Elevator` ha bisogno di un'operazione `summonElevator` (chiama l'ascensore). L'espressione "segnala all'ascensore di spostarsi" relativa alla classe `ElevatorButton` comporta che la classe `Elevator` deve fornire un servizio "corsa". Prima che l'ascensore possa spostarsi, comunque, deve chiudere la porta: la scelta più appropriata sembra essere quella di aggiungere l'operazione `prepareToLeave` (preparazione alla corsa) nella classe `Elevator`, durante la quale l'ascensore *effettua le operazioni preliminari necessarie prima di iniziare una corsa. Le espressioni* della classe `Door` implicano che la porta invii un messaggio alla persona indicandole di uscire dall'ascensore o di entrarvi. A questo scopo, creiamo due operazioni nella classe `Person` che chiamiamo `exitElevator` (esci dall'ascensore) ed `enterElevator` (entra nell'ascensore).

Per ora possiamo evitare di preoccuparci eccessivamente dei parametri e dei tipi restituiti, perché stiamo soltanto cercando di acquisire una conoscenza elementare delle operazioni di ciascuna classe. Proseguendo con la progettazione, potrà capitare che il numero di operazioni di qualche classe possa variare, perché potremmo volerne implementare delle nuove o eliminarne alcune non strettamente necessarie.

4.10.1 I diagrammi di sequenza

Il *diagramma di sequenza* UML descrive il modo in cui gli oggetti si scambiano i messaggi durante la simulazione. Possiamo utilizzarlo nel nostro progetto (Figura 4.26) per rappresentare il "ciclo di simulazione", ovvero i passi che l'edificio dovrà ripetere per tutta la durata della simulazione. Nel diagramma, ogni oggetto è rappresentato da un rettangolo (posto in alto) che ne contiene il nome. Scriveremo i nomi degli oggetti nel diagramma di sequenza seguendo la convenzione che abbiamo introdotto per il diagramma degli oggetti, nella Sezione "Pensare in termini di oggetti" del Capitolo 2 (Figura 2.45). La linea tratteggiata che parte dal rettangolo che rappresenta un oggetto e si estende verso il basso è la *linea di evoluzione* dell'oggetto, e rappresenta la progressione del suo stato e delle attività che esso compie nel tempo. Le azioni si verificano lungo la linea di evoluzione in ordine cronologico, a partire dalla sommità verso la base del diagramma: ciò significa che un'azione che si trova vicino all'inizio della linea di evoluzione si verifica prima di un'azione che si trova più in basso. Un messaggio scambiato tra due oggetti si rappresenta con una linea e una freccia piena che parte dall'oggetto che invia il messaggio e punta alla linea di evoluzione dell'oggetto che lo riceve. Il messaggio ha come effetto quello di invocare l'operazione corrispondente nell'oggetto ricevente. Il nome del messaggio si scrive sopra la linea del messaggio e include gli eventuali parametri passati. Per esempio l'oggetto della classe `Building` invia il messaggio `processTime` all'oggetto della classe `Elevator`. Il nome del messaggio compare sulla linea corrispondente e il nome del parametro (`currentTime`) tra le parentesi a destra del messaggio; ogni parametro è seguito da un segno di due punti e dal suo tipo.

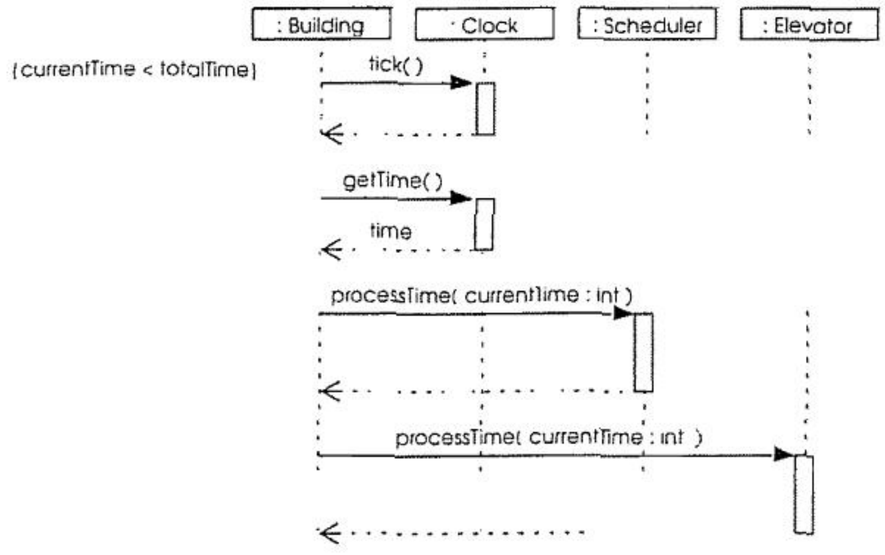


Figura 4.26 Diagramma di sequenza che rappresenta il ciclo di simulazione.

Se un oggetto restituisce il flusso di controllo o restituisce un valore, il messaggio di ritorno è rappresentato da una linea tratteggiata con una freccia e si estende da tale oggetto all'oggetto che aveva inviato originariamente il messaggio. Per esempio l'oggetto della classe `Clock` restituisce `time` in risposta al messaggio `getTime` ricevuto dall'oggetto della clas-

se `Building`. I rettangoli che si trovano lungo le linee di evoluzione degli oggetti sono detti *attivazioni* e rappresentano la durata di un'attività. Un'attivazione inizia quando un oggetto riceve un messaggio e l'altezza del rettangolo corrisponde alla durata dell'attività o delle attività iniziate dal messaggio: a durata maggiore corrisponde un rettangolo più alto.

Il testo all'estrema sinistra del diagramma in Figura 4.26 indica un vincolo temporale. Esso indica che, finché il tempo trascorso è minore dell'intera durata della simulazione (`currentTime < totalTime`), gli oggetti continuano a inviarsi messaggi nella sequenza rappresentata dal diagramma. La Figura 4.27 rappresenta il modo in cui il meccanismo di gestione degli arrivi gestisce il tempo e crea nuove persone ai piani. In questo diagramma assumiamo che esso faccia arrivare una persona su ogni piano all'orario passatogli dall'edificio. Seguiamo il flusso dei messaggi attraverso il diagramma di sequenza.

L'oggetto `building` fa partire la sequenza inviando il messaggio `processTime` all'oggetto `scheduler` e passandogli l'orario corrente. L'oggetto `scheduler` deve quindi decidere se creare una nuova persona al primo piano (rappresentato dall'oggetto `floor1` della classe `Floor`), e per fare ciòesso deve prima verificare che il piano sia libero (questo vincolo fa parte della definizione del problema). L'oggetto `scheduler` invia a questo scopo un messaggio `isOccupied` all'oggetto `floor1` e, quest'ultimo restituisce un valore di tipo `bool`, come indica la linea tratteggiata del messaggio di ritorno. A questo punto la linea di evoluzione dell'oggetto `scheduler` si sdoppia in due possibili evoluzioni parallele che rappresentano ogni possibile sequenza di messaggi che l'oggetto può inviare sulla base del valore restituito dall'oggetto `floor1`. La linea di evoluzione di un oggetto si può suddividere in più linee di vita per indicare l'*esecuzione condizionale delle attività*: per ognuna delle suddivisioni deve essere presente una condizione e le linee di evoluzione risultanti corrono parallele a quella principale (eventualmente per convergere nuovamente in seguito).

Se l'oggetto `floor1` restituisce `true` (cioè il piano è occupato) l'oggetto `scheduler` chiama la propria funzione `delayArrival` (ritarda l'arrivo), passandole un parametro che indica la riprogrammazione dell'orario di arrivo di una persona al primo piano. Questa funzione non è un'operazione della classe `Scheduler`, non essendo invocata da un altro oggetto: essa è semplicemente un'attività che la classe `Scheduler` effettua durante un'operazione. Notate che, quando l'oggetto `scheduler` invia un messaggio a se stesso (invoca una propria funzione membro), la barra di attivazione del messaggio è centrata sul lato destro della barra di attivazione corrente. Se, invece, l'oggetto `floor1` restituisce `false` (il piano è libero), l'oggetto `scheduler` crea un nuovo oggetto di tipo `Person`. Quando viene creato un nuovo oggetto, il suo rettangolo nel diagramma di sequenza si pone in una posizione verticale che corrisponde all'istante in cui è stato creato l'oggetto. Un oggetto che crea un altro oggetto invia un messaggio con la dicitura "create" racchiusa tra i simboli « e ». La freccia di questo messaggio punta al rettangolo del nuovo oggetto. Una "X" al termine della linea di vita di un oggetto denota la sua distruzione. [Nota: Il nostro diagramma di sequenza non rappresenta la distruzione degli oggetti della classe `Person`, per cui non ci sono segni "X". La creazione e la distruzione dinamica degli oggetti con gli operatori del C++ `new` e `delete` è illustrata nel Capitolo 7].

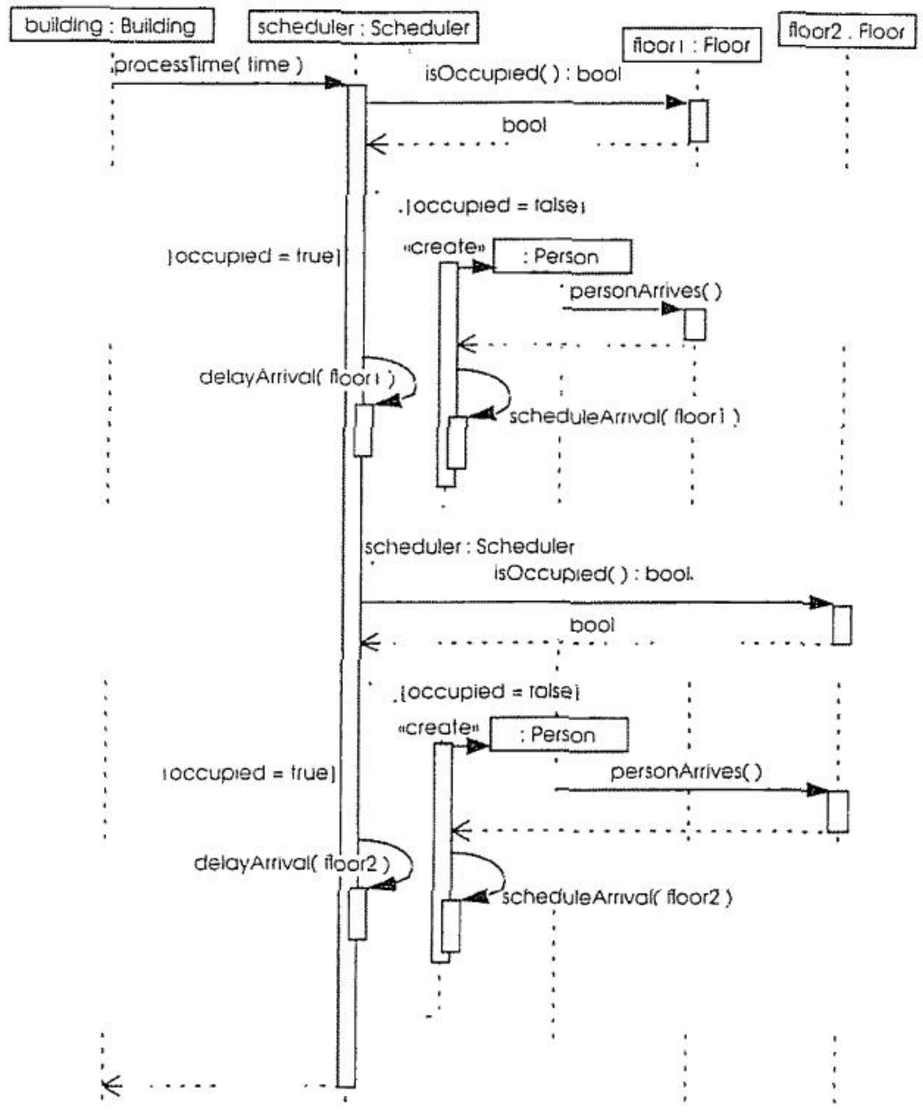


Figura 4.27 Diagramma di sequenza per il processo di programmazione.

Dopo la creazione di un nuovo oggetto `Person`, questo deve accedere al primo piano. Il nuovo oggetto `Person` quindi invia un messaggio `personArrives` all'oggetto `floor1` notificandogli l'arrivo di una persona. Non appena l'oggetto `scheduler` ha creato un nuovo oggetto della classe `Person`, programma l'arrivo di una nuova persona al primo piano. L'oggetto `scheduler` invoca la propria funzione `scheduleArrival`, e come vedere la barra di attivazione di questa chiamata e centrata sul margine destro della barra di attivazione corrente. La funzione `scheduleArrival` non è un'operazione, ma un'attività che la classe `Scheduler` effettua all'interno di un'operazione. A questo punto le due linee di vita conver-

gono. L'oggetto `scheduler` gestisce il secondo piano in maniera analoga al primo. Quando il meccanismo di gestione degli arrivi ha finito anche la gestione del secondo piano, esso restituisce il controllo a `building`.

In questa sezione abbiamo discusso delle operazioni delle classi e abbiamo introdotto il diagramma di sequenza UML. Nella sezione "Pensare in termini di oggetti" del Capitolo 5 esamineremo il modo in cui gli oggetti di un sistema interagiscono tra loro per portare a termine un compito specifico, e cominceremo a implementare il nostro simulatore in C++.

Esercizi di autovalutazione

4.1 Completate le seguenti affermazioni:

a) Le liste e le tabelle di valori sono memorizzate in _____.
b) Gli elementi di un array sono correlati tra loro perché hanno lo stesso_____ e lo stesso _____
c) Il numero che si riferisce ad un elemento particolare di un array è detto _____
d) Conviene utilizzare una _____ per dichiarare la dimensione di un array, perché in questo modo il programma risulta più scalabile.
e) Il procedimento con cui si mettono gli elementi di un array in un determinato ordine si chiama _____ dell'array.
f) Il procedimento con cui si determina se un array contiene un determinato valore chiave si chiama _____.
g) Un array che utilizza due indici si dice array _____

4.2 Stabilite se le seguenti affermazioni sono vere o false. Per quelle false, datene una breve spiegazione.

a) Un array può memorizzare molti tipi di valori diversi.
b) L'indice di un array dovrebbe essere generalmente di tipo `float`.
c) Se una lista di inizializzatori ne contiene meno del numero di elementi dell'array, i restanti elementi sono inizializzati all'ultimo valore della lista.
d) Se una lista di inizializzatori contiene più valori del numero di elementi dell'array, avrete un errore.
e) Se passate un singolo elemento di un array a una funzione, e questa lo modifica, da quel momento in poi l'array conterrà l'elemento modificato.

4.3 Questo esercizio si basa sull'array `fractions`.

a) Definite una variabile a sola lettura inizializzata a 10.
b) Dichiarate un array con `arraySize` elementi di tipo `float` e inizializzate gli elementi a 0.
c) Leggete il valore del quarto elemento dell'array, a partire dall'inizio dell'array..
d) Riferite l'elemento 4.
e) Assegnate il valore `1.667` all'elemento 9.
f) Assegnate il valore `3.333` all'ultimo elemento dell'array.
g) Visualizzate gli elementi 6 e 9 con due cifre di precisione e controllate l'output effettivamente presente sullo schermo.
h) Visualizzate tutti gli elementi dell'array utilizzando un costrutto `for`. Definite la variabile intera `x` come variabile di controllo del ciclo. Scrivete l'output prodotto.

4.4 Questo esercizio si basa sull'array `table`.

a) Dichiarate `table` come array di interi con 3 righe e 3 colonne. Per ipotesi, supponiamo che la variabile a sola lettura `arraySize` abbia un valore pari a 3.
b) Quanti elementi contiene l'array?

c) Utilizzate un costrutto `for` per inizializzare ciascun elemento dell'array con la somma dei suoi indici. Come variabili di controllo utilizzate le due variabili intere `x` e `y`.
d) Scrivete un segmento di programma per visualizzare i valori presenti in ogni elemento dell'array `table` in un tabella di 3 righe e 3 colonne. Supponiamo che l'array sia stato inizializzato con la dichiarazione

```
int table[ arraySize ][ arraySize ] =
   { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

Utilizzate come variabili di controllo le due variabili intere x e y. Scrivete l'output prodotto.

4.5 Trovate l'errore presente in ogni segmento di programma che vi proponiamo, e cercate di correggerlo.

a)
```
#include <iostream.h>;
```
b)
```
arraySize = 10;  // arraySize è const
```
c) Supponendo
```
int b[ 10 ] = { 0 };
for ( int a = 0; a <= 10; a++ )
   b[ a ] = 1;
```
d) Supponendo
```
int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
a[ 1, 1 ] = 5;
```

Risposte agli esercizi di autovalutazione

4.1 a) Array. b).Nome, tipo. c) Indice. d) Variabile a sola lettura. e) Ordinamento. f) Ricerca. g) Bidimensionale, a due indici o matrice.

4.2 a) Falsa. Un array contiene solo elementi dello stesso tipo. b) Falsa. L'indice di un array normalmente è un intero o un'espressione intera. c) Falsa. I restanti elementi sono inizializzati automaticamente a zero. d)Vera. e) Falsa. I singoli elementi di un array sono passati per valore. Se invece viene passato un intero array a una funzione, tutte le modifiche verranno effettuate sull'array originario.

4.3 a)
```
const int arraySize = 10;
```
b)
```
float fractions[ arraySize ] = { 0 };
```
c)
```
fractions[ 3 ]
```
d)
```
fractions[ 4 ]
```
e)
```
fractions[ 9 ] = 1.667;
```
f)
```
fractions[ 6 ] = 3.333;
```
g)
```
cout  << setiosflags( ios::fixed | ios::showpoint )
      << setprecision( 2 ) << fractions[ 6 ] << ' '
      << fractions[ 9 ] << endl;
```

Output: 3.33 1.67.

h)
```
for ( int x = 0; x < arraySize; x++ )
   cout << "fractions[" << x << "] = " << fractions[ x ]
        << endl;
```

```
fractions[ 0 ] = 0
fractions[ 1 ] = 0
fractions[ 2 ] = 0
fractions[ 3 ] = 0
fractions[ 4 ] = 0
fractions[ 5 ] = 0
fractions[ 6 ] = 2.333
fractions[ 7 ] = 0
fractions[ 8 ] = 0
fractions[ 9 ] = 1.667
```

4.4 a) `int table[ arraySize ][ arraySize ];`

b) Nove.

c)
```
for ( x = 0; x < arraySize; x++ )
   for ( y = 0; y < arraySize; y++ )
      table[ x ][ y ] = x + y;
```

d)
```
cout << "      [0]  [1]  [2]" << endl;
for ( int x = 0; x < arraySize; x++ ) {
   cout << '[' << x << "] ";
   for ( int y = 0; y < arraySize; y++ )
      cout << setw( 3 ) << table[ x ][ y ] << "  ";
   cout << endl;
}
```

Output:

```
     [0]  [1]  [2]
[0]   1    8    0
[1]   2    4    6
[2]   5    0    0
```

4.5 a) Errore: Il punto e virgola che termina della direttiva al preprocessore `#include`.
Correzione: Eliminate il punto e virgola.

b) Errore: Si assegna un valore a una variabile a sola lettura.
Correzione: Assegnate il valore alla variabile a sola lettura durante la sua dichiarazione.

c) Errore: Si referenzia un elemento dell'array oltre i limiti consentiti (`b[10]`).
Correzione: Ponete 9 come valore finale della variabile di controllo.

d) Errore: Si indicizza l'array in modo scorretto.
Correzione: Modificate l'istruzione in `a[ 1 ][ 1 ] = 5;`

Esercizi

4.6 Completate le seguenti affermazioni:

a) Il C++ memorizza le liste di valori in _____

b) Gli elementi di un array sono correlati per il fatto che _____

c) Quando referenziate un elemento di un array, il numero posizionale racchiuso tra le parentesi si chiama _____.

d) I nomi dei quattro elementi dell'array p sono _____, _____, _____ e _____

e) Specificare il nome di un array, stabilirne il tipo e specificare il numero di elementi che contiene è un procedimento che prende il nome di _____ dell'array.

f) Il procedimento con cui si pongono gli elementi di un array in ordine crescente o decrescente si chiama _____

g) In un array bidimensionale, per convenzione il primo indice identifica la _____ di un elemento, mentre il secondo identifica la _____ di un elemento.

h) Un array mxn contiene _____ righe, _____ colonne e _____ elementi.

i) Il nome dell'elemento nella riga 3 e nella colonna 5 dell'array d è _____

4.7 Stabilite quali affermazioni sono vere e quali sono false. Per quelle false, [illegible] una breve spiegazione.

a) Per riferire una locazione o un elemento particolare di un array, basta specificare il nome dell'array e il valore dell'elemento desiderato.

b) La dichiarazione di un array riserva dello spazio in memoria per l'array.

c) Per indicare al compilatore di riservare 100 locazioni per l'array di interi p, bisogna scrivere la dichiarazione

`p[ 100 ];`

d) Un programma che inizializza a zero tutti i 15 elementi di un array di interi deve contenere almeno un inizializzatore per istruzione.

e) Un programma che calcola la somma di tutti gli elementi di un array bidimensionale deve contenere due costrutti `for` nidificati.

4.8 Scrivete un'istruzione per ognuno dei compiti che vi proponiamo:

a) Visualizzate il valore del settimo elemento dell'array `f`.

b) Trasferite un valore letto in input nell'elemento 4 dell'array `b`, che è un array unidimensionale di valori a virgola mobile.

c) Inizializzate tutti i 5 elementi dell'array unidimensionale `g` a 8.

d) Calcolate e visualizzate la somma dei 100 elementi dell'array `c` di valori a virgola mobile.

e) Copiate l'array `a` nella prima porzione dell'array `b`. Supponete che siano stati dichiarati con `float a[ 11 ], b[ 34 ];`

f) Determinate e visualizzate il più piccolo e il più grande valore contenuto nell'array `w`, che è un array unidimensionale di valori a virgola mobile.

4.9 Questo esercizio si basa sull'array `t`, che è un array di interi 2x3.

a) Dichiarate l'array `t`.

b) Quante righe ha `t`?

c) Quante colonne ha `t`?

d) Quanti elementi ha `t`?

e) Scrivete il nome di tutti gli elementi della seconda riga di `t`.

f) Scrivete il nome di tutti gli elementi della terza colonna di `t`.

g) Scrivete una sola istruzione che azzeri l'elemento in riga 1 colonna 2.

h) Scrivete una serie di istruzioni che inizializzano a zero tutti gli elementi di `t`. Non utilizzate una struttura di iterazione.

i) Scrivete un costrutto `for` nidificato che azzera tutti gli elementi di `t`.

j) Scrivete un'istruzione che legge in input i valori degli elementi di `t`.

k) Scrivete una serie di istruzioni che determina e visualizza il valore più piccolo contenuto nell'array `t`.

l) Scrivete un'istruzione che visualizza gli elementi della prima riga di `t`.

m) Scrivete un'istruzione che somma tutti gli elementi della quarta colonna di `t`.

n) Scrivete una serie di istruzioni che visualizza l'array `t` in formato tabulare. Includete gli indici di colonna come intestazione della tabella, e gli indici di riga a sinistra di ogni riga.

4.10 Risolvete il seguente problema utilizzando un array unidimensionale. Una compagnia paga i propri rappresentanti con commissioni. I rappresentanti prendono $200 a settimana più il 9% di commissione sulle vendite settimanali. Per esempio, un rappresentante che vende per un valore di $5000 prende $200 più il 9% di $5000, per un totale di $650. Utilizzando un array di contatori, scrivete un programma che determina il numero di rappresentanti appartenenti a queste classi di retribuzione (troncate sempre il salario all'intero più vicino):

a) $200-$299

b) $300-$399

c) $400-$499

d) $500-$599

e) $600-$699

f) $700-$799

g) $800-$899

h) $900-$999

i) $1000 e oltre

4.11 Il *bubble sort* presentato in Figura 4.16 è inefficiente su array di grosse dimensioni. Modificatelo come segue per migliorarne l'efficienza.

a) Dopo il primo passaggio, siamo certi che il numero più grande si trovi nell'ultimo elemento dell'array; dopo il secondo passaggio, i due numeri più grandi si trovano "al posto giusto", e così via. Modificate il programma in modo che effettui solo otto confronti al secondo passaggio (anziché nove), sette al terzo (anziché otto) e così via.

b) I dati dell'array possono già essere in ordine, o molti vicini all'ordine desiderato, per cui non c'è ragione di effettuare nove confronti se ne bastano molti meno. Modificate il programma per verificare alla fine di ogni passaggio se sono stati effettuati degli scambi. Se *non* ce ne sono stati, i dati saranno sicuramente già in ordine. Se ce n'è stato almeno uno, si deve procedere con il passaggio successivo.

4.12 Scrivete una sola istruzione per ogni operazione che vi proponiamo:

a) Inizializzate a zero i 10 elementi dell'array di interi `counts`.

b) Aggiungete 1 a ognuno dei 15 elementi dell'array di interi `bonus`.

c) Leggete dalla tastiera 12 valori da trasferire nell'array di valori a virgola mobile `monthlyTemperatures`.

d) Visualizzate i 5 valori dell'array di interi `bestScores` in formato di colonna.

4.13 Trovate l'errore/gli errori nelle seguenti istruzioni:

a) Supponiamo:
```
char str[ 5 ];
cin >> str;  // L'utente digita hello
```

b) Supponiamo:
```
int a[ 3 ];
cout << a[ 1 ] << ' ' << a[ 2 ] << ' ' << a[ 3 ] << endl;
```

c) `float f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`

d) Supponiamo:
```
double d[ 2 ][ 10 ];
d[ 1, 9 ] = 2.345;
```

4.14 Modificate il programma in Figura 4.17 in modo che la funzione `mode` sia in grado di prevedere singolarità per il valore della moda. Modificate anche la funzione `median`, in modo che restituisca la media aritmetica dei due valori centrali, nel caso di un numero pari di elementi.

4.15 Utilizzate un array unidimensionale per risolvere questo problema. Leggete 20 numeri compresi tra 10 e 100, per ogni numero letto, visualizzatelo solo se esso non è stato già immesso in precedenza. Prevedete il "caso peggiore", cioè quando tutti i numeri sono diversi. Utilizzate l'array più piccolo possibile per risolvere il problema.

4.16 Dite in che ordine sono azzerati gli elementi di `sales`, array bidimensionale 3x5, in questo segmento di programma:
```
for ( row = 0; row < 3; row++ )
   for ( column = 0; column < 5; column++ )
      sales[ row ][ column ] = 0;
```

4.17 Scrivete un programma che simula il lancio di due dadi. Il programma dovrebbe utilizzare `rand` per lanciare il primo dado e successivamente il secondo. Dopo il lancio dei dadi il programma deve calcolare la somma dei due valori. *Nota:* dato che ogni dado può dare un valore intero tra 1 e 6, la somma varierà tra 2 e 12, con valore più probabile uguale a 7 e i due valori meno probabili uguali a 2 e 12.
La Figura 4.28 illustra le 36 possibile combinazioni dei due dadi. Il programma deve lanciare i due dadi per 36000 volte.
Utilizzate un array unidimensionale per annotare tutte le occorrenze di ogni possibile somma.

Visualizzate il risultato *in formato tabulare*.
Determinate anche se i totali sono verosimili, cioè se ci sono realmente 6 modi per avere 7, e quindi se all'incirca un sesto di tutti i lanci dà 7.

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figura 4.28 I 36 possibili risultati del lancio di due dadi.

4.18 Che cosa fa questo programma?
```
1  // ex04_18.cpp
2  #include <iostream.h>
3
4  int whatIsThis( int [], int );
5
6  int main()
7  {
8     const int arraySize = 10;
9     int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11    int result = whatIsThis( a, arraySize );
12
13    cout << "Result is " << result << endl;
14    return 0;
15 }
16
17 int whatIsThis( int b[], int size )
18 {
19    if ( size == 1 )
20       return b[ 0 ];
21    else
22       return b[ size - 1 ] + whatIsThis( b, size - 1 );
23 }
```

4.19 Scrivete un programma che esegue 1000 giocate di crap e risponde alle seguenti domande:

a) Quante giocate sono vinte al primo lancio, secondo lancio, ..., ventesimo lancio e dopo il ventesimo lancio?

b) Quante giocate sono perse al primo lancio, secondo lancio, ..., ventesimo lancio e dopo il ventesimo lancio?

c) Che possibilità si hanno di vincere? (Nota: per farvi un'idea, questo è uno dei giochi più generosi nei casinò. Riuscite a capire che significa?)

d) Quanto dura in media una giocata?

e) Man mano che il gioco prosegue, aumentano le probabilità di vittoria?

4.20 (*Sistema di prenotazione di una compagnia aerea*) Una piccola compagnia aerea ha appena acquistato un computer da dedicare al sistema di prenotazione automatica dei voli. La compagnia vi ha chiesto di programmare il sistema. Il vostro compito è scrivere un programma che assegna i posti di ogni volo dell'unico aereo posseduto dalla compagnia (10 posti in tutto).
Il vostro programma dovrebbe visualizzare questo menu di opzioni: `Digitare 1 per "fumatori"` e `Digitare 2 per "non fumatori"`. Se la persona digita 1, il programma le assegna un posto nel reparto fumatori (posti 1-5). Se invece digita 2, il posto si troverà nel reparto non fumatori (posti 6-10). Il programma deve visualizzare una carta di imbarco che indica il numero del posto della persona e se si trova nel reparto fumatori o non fumatori dell'aereo.
Utilizzate un array unidimensionale per rappresentare i posti dell'aereo. Inizializzate tutti gli elementi dell'array a 0, per indicare che all'inizio tutti i posti sono vuoti. Man mano che vengono assegnati, impostate i posti occupati a 1.
Il vostro programma ovviamente non deve riassegnare un posto già occupato. Quando il reparto fumatori si è riempito, chiedete al passeggero se è accettabile andare nel reparto non fumatori (e viceversa). Se sì, assegnategli un posto nell'altro reparto. Se no, visualizzate `"Il prossimo volo parte tra 3 ore"`

4.21 Che cosa fa questo programma?

```
1   // ex04_21.cpp
2   #include <iostream.h>
3
4   void someFunction( int [], int );
5
6   int main()
7   {
8      const int arraySize = 10;
9      int a[ arraySize ] = {32,27,64,18,95,14,90,70,60,37};
10
11     cout << "The values in the array are:" << endl;
12     someFunction( a, arraySize );
13     cout << endl;
14     return 0;
15  }
16
17  void someFunction( int b[], int size )
18  {
19     if ( size > 0 ) {
20        someFunction( &b[ 1 ], size - 1 );
21        cout << b[ 0 ] << " ";
22     }
23  }
```

4.22 Risolvete il seguente problema utilizzando un array bidimensionale. Una compagnia ha quattro rappresentanti (da 1 a 4) che vendono cinque prodotti diversi (da 1 a 5). Una volta al giorno ogni rappresentante sottoscrive una bolla per ogni tipo di prodotto in vendita. Ogni bolla contiene:

a) Il numero di identificazione del rappresentante
b) Il numero del prodotto
c) Il valore del venduto per il prodotto in questione durante la giornata

Ogni rappresentante quindi sottoscrive da 0 a 5 bolle al giorno. Supponete di aver accesso a tutte le bolle del mese scorso. Scrivete un programma che legge le informazioni di tutte le vendite del mese scorso e riepiloga le vendite di ogni rappresentante. Tutti i totali vanno memorizzati nell'array bidimensionale `sales`. Dopo l'elaborazione dei dati visualizzate i risultati in una tabella, in cui ogni colonna corrisponde a un rappresentante diverso e ogni riga corrisponde a un prodotto diverso. Sommate trasversalmente ogni riga per ottenere le vendite totali di ciascun prodotto nel mese scorso. Sommate trasversalmente ogni colonna per ottenere il valore del venduto di ogni rappresentante. Quando visualizzate i risultati includete questi totali trasversali e i totali di riga e di colonna.

4.23 (*Turtle graphics*) Il linguaggio Logo, particolarmente utilizzato sui personal computer, ha dato la fama al concetto di *turtle graphics*, disegnare con le tartarughe. Immaginatevi una tartaruga meccanica che si fa il giro della vostra stanza, controllata da un programma in C++. La tartaruga mantiene una penna in due posizioni, alzata e abbassata. Quando la penna è abbassata, man mano che la tartaruga si sposta traccia un disegno, cosa che non succede ovviamente se la penna è tenuta alzata. In questo esercizio proveremo a simulare la tartaruga, creando uno schizzo elettronico.
Utilizzate l'array 20x20 `floor`, che rappresenta il pavimento, inizializzando i suoi elementi a zero. Leggete i comandi da un array che li contiene. Tenete traccia della posizione corrente della tartaruga in ogni momento, annotando se la penna è mantenuta alzata o abbassata. Supporremo che la tartaruga inizi il suo cammino dalla posizione 0,0 del pavimento, con la penna alzata. Ecco il set di comandi-tartaruga che il vostro programma deve riconoscere:

Comando	Significato
1	Alzare la penna
2	Abbassare la penna
3	Girare a destra
4	Girare a sinistra
5, n	Spostarsi in avanti di n spazi
6	Visualizzare l'array 20x20
9	Fine dei dati (sentinella)

Supponiamo che la tartaruga si trovi nelle vicinanze del centro del pavimento. Il seguente "programma" disegna un quadrato 12x12, terminando con la penna alzata:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

Se la tartaruga procede con la penna abbassata, impostate gli elementi del pavimento su cui passa a 1. Quando viene dato il comando 6 (visualizzazione), per ogni elemento dell'array visualizzate un asterisco se vale 1, uno spazio se vale 0. Se volete, potete aggiungere altri comandi al set che vi abbiamo fornito noi, per accrescere le potenzialità della vostra tartaruga.

4.24 (*Il giro del cavallo*) Gli scacchisti più accaniti conosceranno già questo problema, proposto originariamente dal matematico Eulero. La domanda è: Il cavallo può muoversi su una scacchiera vuota toccando ciascuno dei 64 quadrati una e una sola volta? Proviamo a studiare questo problema in questo esercizio.
La mossa del cavallo assomiglia a una lettera L, perché avanza di due caselle in una direzione e di una nella direzione perpendicolare.

Partendo da una casella al centro della scacchiera, il cavallo può effettuare otto mosse diverse, che numeriamo da 0 a 7, come in Figura 4.29.

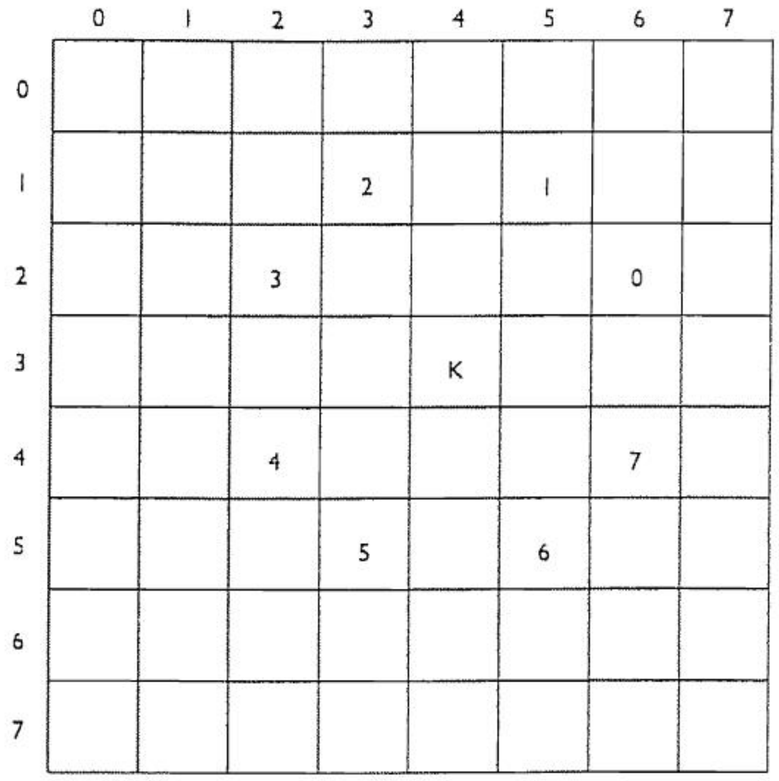


Figura 4.29 Le otto possibili mosse del cavallo.

a) Disegnate una scacchiera 8x8 su un foglio di carta e cercate di effettuare manualmente il giro del cavallo. Scrivete un 1 nel primo quadrato in cui vi spostate, un 2 nel secondo, un 3 nel terzo e così via. Prima di cominciare il giro, fate una stima di dove pensate di arrivare, ricordando che un giro completo tocca 64 caselle. Bene, ora datevi da fare. Dove siete arrivati? La vostra stima era azzeccata?

b) Bene, ora cerchiamo di scrivere un programma per risolvere il problema del giro del cavallo. La scacchiera è rappresentata dall'array bidimensionaledi dimensione 8x8 `board`. Ogni casella è inizialmente azzerata. Descriviamo ognuna delle 8 mosse in termini dei suoi componenti orizzontali e verticali. Per esempio, una mossa di tipo 0 è mostrata in Figura 4.29 e consiste in due caselle in senso orizzontale a destra e una casella in senso verticale in alto. La mossa 2 consiste in una casella in senso orizzontale a sinistra e due caselle in senso verticale in alto. Le mosse in senso orizzontale a sinistra e in senso orizzontale in alto sono indicate da numeri negativi. Le otto mosse possono essere descritte da due array unidimensionali, `horizontal` e `vertical`, come segue:

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
```



```
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -2
vertical[ 1 ] = -2
vertical[ 2 ] = -1
vertical[ 3 ] = 1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

Le variabili `currentRow` e `currentColumn` indicano la riga e la colonna in cui si trova correntemente il cavallo. Per effettuare una mossa del tipo `moveNumber`, dove `moveNumber` è compreso tra 0 e 7, il vostro programma utilizzerà le istruzioni

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Prevedete un contatore da far variare tra 1 e 64. Registrate l'ultimo valore assunto dal contatore nell'ultima casella toccata dal cavallo. Ricordate di verificare ogni mossa per controllare se il cavallo ha già toccato quella casella. Evitate anche di far precipitare il cavallo fuori dalla scacchiera. Bene, ora siete pronti per scrivere il programma. Lanciatelo e dite quante mosse valide avete ottenuto.

c) Dopo il vostro primo tentativo di risolvere il problema del giro del cavallo, avrete probabilmente notato alcuni particolari, come per esempio delle mosse da evitare. Utilizzate il vostro intuito nell'euristica di questo problema, o in altri termini per una strategia risolutiva. L'euristica non garantisce il risultato, ma è un sesto senso che aiuta a risolvere i problemi. Avrete notato che le caselle più esterne sono più problematiche di quelle al centro della scacchiera. Infatti le caselle più diaboliche sono proprio le quattro caselle agli angoli.
Andando a intuito, potreste pensare di provare a piazzare il cavallo prima in queste caselle difficili, rimandando quelle più facili a un secondo tempo, quando la scacchiera sarà più congestionata.
Possiamo sviluppare un'*euristica dell'accessibilità*, classificando ogni casella a seconda della facilità di accedere a essa. Sposteremo poi il cavallo sempre sulle caselle meno accessibili, se ce lo consente ovviamente almeno una delle otto mosse possibili del cavallo. Definiamo l'array bidimensionale `accessibility`, che contiene dei numeri che indicano da quante caselle è possibile accedere a ognuna. Su una scacchiera vuota, ogni casella centrale varrà 8, ogni casella ad angolo varrà 2 e le altre varranno 3, 4 o 6, come segue:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Ora scrivete una versione del programma che utilizzi l'euristica dell'accessibilità. In ogni momento dovete cercare di spostare il cavallo nella casella che ha accessibilità minore. Nel caso di più caselle che hanno valori uguali di accessibilità, il cavallo può spostarsi su una qualsiasi di esse. Il giro può quindi iniziare da una qualsiasi delle quattro caselle ad angolo. Notate che man mano che il cavallo si sposta nella scacchiera, il vostro programma dovrebbe ridurre il valore dell'accessibilità delle caselle, perché molte risulteranno occupate. In

questo modo, avrete in ogni momento l'accessibilità effettiva di ogni casella, uguale al numero preciso di caselle da cui potrete raggiungerla. Eseguite il programma che riuscite a scrivere. Ce l'avete fatta a effettuare un giro completo? Modificate il programma in modo che effettui 64 tentativi di giri completi, ognuno dei quali parte da una casella diversa della scacchiera. Quanti giri completi riuscite a ottenere?

d) Scrivete una versione del programma che quando incontra più caselle con la stessa accessibilità, guarda oltre e vede quante ne sono raggiungibili da ciascuna di esse. Il vostro programma dovrebbe decidere di spostarsi sulla casella da cui si arriva a un'altra con accessibilità bassa.

4.25 (*Il giro del cavallo: vari approcci di forza bruta*) Nell'Esercizio 4.24 abbiamo sviluppato una soluzione per questo problema. L'approccio euristico che abbiamo utilizzato è in grado di generare molte soluzioni in modo ottimale. La velocità dei computer aumenta ogni giorno, per cui possiamo anche cercare di risolvere i problemi con algoritmi meno efficienti e meno sofisticati. A noi piace chiamare questo tipo di strategie risolutive "approcci di forza bruta".

a) Utilizzate i numeri casuali per spostare il cavallo su caselle casuali della scacchiera, sempre lungo le mosse possibili. Il vostro programma dovrebbe tentare un giro completo e visualizzare la scacchiera. Dove riuscite ad arrivare con questo approccio?

b) Con ogni probabilità l'approccio a) non vi ha portato molto lontano. Modificate il programma per effettuare 1000 tentativi. Utilizzate un array unidimensionale per tener traccia del numero di giri di ogni lunghezza. Al termine dei 1000 tentativi, il programma deve visualizzare queste informazioni in una tabella. Qual è stato il miglior risultato?

c) Ancora una volta, l'approccio b) vi avrà regalato qualche giro degno di nota, ma difficilmente avrete avuto un solo tour completo. Adesso lasciatelo andare a briglia sciolta, in modo che effettui tentativi su tentativi finché non riesce a effettuare un giro completo. Fate attenzione comunque: questa versione del programma può anche girare per ore intere. Anche qui stilate una tabella con il numero di giri di ogni lunghezza. Quanti tour incompleti ha generato il programma prima del tour completo? Quanto tempo ha impiegato?

d) Confrontate la versione "forza bruta" con la versione "euristica". Quale di esse richiede uno studio più attento? Quale algoritmo è stato implementato con maggiore difficoltà? Quale di essi richiede un computer più potente? Possiamo essere certi dall'inizio di ottenere un giro completo con l'approccio euristico? E con l'approccio di forza bruta? Argomentate i pro e i contro dei due tipi di approccio.

4.26 (*Problema delle otto regine*) Un'altra chicca per gli amanti degli scacchi è il problema delle otto regine. In breve: è possibile sistemare otto regine su una scacchiera vuota in modo che nessuna possa attaccarne un'altra, cioè in modo tale che nessuna coppia di regine si trovi sulla stessa riga, colonna o diagonale? Fate uso dei ragionamenti che abbiamo sviluppato nell'Esercizio 4.24 per formulare un'euristica di questo problema. Suggerimento: è possibile assegnare a ciascuna casella un valore che indica quante caselle della scacchiera devono essere scartate se una regina viene piazzata in quella casella. Gli angoli avrebbero un valore di 22, come in Figura 4.30. Dopo aver messo in ogni casella questi numeri di "scarto", una buona euristica può essere la seguente: piazzare la regina successiva nella casella che ha numero di scarto minore. Perché questo tipo di strategia risulta intuitivamente intrigante?

Figura 4.30 Le 22 caselle da scartare quando si pone una regina nell'angolo a sinistra in alto.

4.27 (*Problema delle otto regine: vari approcci di forza bruta*) In questo esercizio svilupperete diversi approcci di forza bruta al problema dell'Esercizio 4.26.

a) Risolvete il problema delle otto regine con l'approccio dei numeri casuali che abbiamo introdotto nell'Esercizio 4.25.

b) Utilizzate una tecnica esaustiva, piazzando le regine in tutte le possibili combinazione sulla scacchiera.

c) Avete dei motivi per dubitare che l'approccio di forza bruta possa esservi di qualche aiuto in questo problema? Quali sono?

d) Mettete a confronto l'approccio dei numeri casuali con l'approccio esaustivo, indicandone i pro e i contro.

4.28 (*Il giro del cavallo: test di chiusura del giro*) Nel giro del cavallo, si ha un giro completo quando il cavallo ha toccato tutte le 64 caselle della scacchiera una e una sola volta. Il giro si dice chiuso quando la 64-esima casella dista di una mossa dalla casella di partenza del giro. Modificate il programma che avete scritto nell'Esercizio 4.24 per verificare se il giro effettuato è anche chiuso.

4.29 (*Il crivello di Eratostene*) Un numero primo è un intero divisibile soltanto per 1 e per se stesso. Il crivello di Eratostene è un metodo per trovare i numeri primi. Funziona così:

a) Create un array con tutti gli elementi inizializzati a 1 (`true`). Gli elementi dell'array che hanno per indici dei numeri primi resteranno uguali a 1. Tutti gli altri invece verranno impostati a zero.

b) Partite dall'indice 2 (l'indice 1 è necessariamente primo), e ogni volta che trovate un elemento dell'array uguale a 1, impostate a zero tutti gli elementi il cui indice è multiplo dell'elemento trovato. Per l'elemento di indice 2, dovremo azzerare tutti gli elementi che si trovano dopo 2 e che sono multipli di due, cioè tutti gli elementi di indice pari; ugualmente per l'elemento di indice 3, dovremo azzerare tutti gli elementi che hanno per indice un suo multiplo, come 6, 9, 12, 15 eccetera, e così via.
Quando avremo completato il procedimento, gli elementi dell'array che valgono ancora 1 avranno per indice un numero primo. Scrivete un programma che implementa il crivello di Eratostene su 1000 elementi, per determinare i numeri primi compresi tra 1 e 999. Ignorate l'elemento 0 dell'array.

4.30 (*Un'altra tecnica di ordinamento: bucket sort*) Questa tecnica di ordinamento prevede l'utilizzo di un array unidimensionale di interi positivi da ordinare, e un array bidimensionale di interi. In questo array le righe hanno indici che vanno da 0 a 9 e le colonne hanno indici che vanno da 0 a n-1, dove n è il numero di valori da ordinare. Ogni riga dell'array bidimensionale è detta *bucket* o *secchiello*. Scrivete la funzione `bucketSort` che riceve come argomenti un array di interi e la sua dimensione ed effettua queste operazioni:

a) Pone ogni valore dell'array unidimensionale in una riga dell'array di bucket, sulla base della cifra delle unità presente nel numero. Per esempio il valore 97 viene posto nella riga 7, 3 nella riga 3 e 100 nella riga 0. Questa si chiama "passata di distribuzione"

b) Con un ciclo sulle righe dell'array bucket copiate i valori nuovamente nell'array originario. Questa si chiama "passata di raccolta". Il nuovo ordine dei valori nell'array unidimensionale è 100, 3 e 97.

c) Ripetete il procedimento per ogni successiva cifra (decine, centinaia, migliaia ecc.).

Alla seconda passata, 100 viene posto nella riga 0, 3 viene posto nella riga 0 (perché 3 non ha la cifra delle decine) e 97 nella riga 9. Dopo la passata di raccolta, l'ordine dei valori nell'array unidimensionale è 100, 3 e 97. Alla terza passata 100 è posto nella riga 1, 3 nella riga 0 e 97 nella riga 0 (dopo il 3). Dopo l'ultima passata di raccolta gli elementi dell'array originale sono in ordine.
Notate che l'array bidimensionale dei bucket è grande dieci volte l'array da ordinare. Questa tecnica di ordinamento è più efficiente del bubble sort, ma richiede molta più memoria. Nel bubble sort lo

spazio richiesto è di un solo dato aggiuntivo. Questo è un esempio di compromesso spazio-tempo: bucket sort usa più memoria di bubble sort, ma è più efficiente. Questa versione di bucket sort copia tutti i dati nuovamente nell'array originario a ogni passata. Un'altra possibilità è quella di creare un secondo array bidimensionale di bucket e scambiare ripetutamente i dati tra i due array di bucket.

Esercizi sulla ricorsione

4.31 (*Ordinamento per selezione*) L'ordinamento per selezione ricerca in un array l'elemento più piccolo. Questo elemento viene poi scambiato di posto con il primo elemento dell'array. Il procedimento si ripete per il sottoarray che inizia dal secondo elemento. Per ogni passata l'array contiene un elemento in più al posto giusto. L'efficienza di questo algoritmo è paragonabile a quella del bubble sort: per un array di *n* elementi occorrono *n* - *1* passate, e per ogni sottoarray occorrono *n* - *1* confronti per trovare l'elemento più piccolo. Quando si arriva al sottoarray che contiene un solo elemento, l'array si considera ordinato. Scrivete la funzione ricorsiva `selectionSort` che implementa questo algoritmo.

4.32 (*Palindromi*) Una stringa palindroma è una stringa che si legge allo stesso modo da sinistra a destra e da destra a sinistra. Per esempio "radar", "able was i ere i saw elba" e (ignorando gli spazi) "a man a plan a canal panama" sono palindromi. Scrivete la funzione ricorsiva `testPalindrome` che restituisce `true` se la stringa memorizzata in un array è un palindromo, altrimenti restituisce `false`. La funzione deve ignorare spazi e segni di punteggiatura.

4.33 (*Ricerca lineare*) Modificate il programma in Figura 4.19 includendo la funzione ricorsiva `linearSearch` che effettua una ricerca lineare nell'array. La funzione deve ricevere come argomenti un array di interi e la sua dimensione. Se viene trovata la chiave di ricerca, la funzione restituisce l'indice corrispondente, altrimenti –1.

4.34 (*Ricerca binaria*) Modificate il programma in Figura 4.20 includendo la funzione ricorsiva `binarySearch` che effettua una ricerca binaria nell'array. La funzione deve ricevere come argomenti un array di interi e gli indici iniziale e finale per la ricerca. Se viene trovata la chiave di ricerca, la funzione restituisce l'indice corrispondente, altrimenti –1.

4.35 (*Le otto regine*) Scrivete una versione ricorsiva del programma delle otto regine dell'Esercizio 4.26.

4.36 (*Visualizzazzione di un array*) Scrivete la funzione ricorsiva `printArray` che riceve come argomenti un array e le sue dimensioni, e che non restituisce alcun dato. La funzione termina il suo lavoro quando riceve un array di dimensione zero.

4.37 (*Visualizzazione di una stringa al contrario*) Scrivete la funzione ricorsiva `stringReverse` che riceve come argomento un array di caratteri, visualizza la stringa al contrario e non restituisce alcun dato. La funzione termina il suo lavoro quando incontra il carattere nullo.

4.38 (*Trovare il valore più piccolo in un array*) Scrivete la funzione ricorsiva `recursiveMinimum` che riceve come argomenti un array di interi e la sua dimensione e restituisce l'elemento più piccolo dell'array. La funzione termina il suo lavoro quando riceve un array di 1 elemento.

CAPITOLO 5

Puntatori e stringhe

Obiettivi

- Imparare a utilizzare i puntatori
- Imparare a passare puntatori alle funzioni simulando la chiamata per riferimento
- Comprendere la stretta correlazione che esiste fra puntatori, array e stringhe
- Comprendere l'utilizzo dei puntatori a funzioni
- Imparare ad utilizzare gli array di stringhe

5.1 Introduzione

In questo capitolo parleremo di una delle caratteristiche più importanti del C++, i puntatori. Si tratta di un aspetto del linguaggio su cui probabilmente tornerete diverse volte prima di acquisirne una certa padronanza. Nel Capitolo 3 abbiamo visto che possiamo eseguire una chiamata per riferimento utilizzando i riferimenti. Grazie ai puntatori potremo ora simulare una chiamata di questo tipo e manipolare strutture di dati dinamiche, cioè dati che possono variare la propria dimensione durante l'esecuzione di un programma: fra queste strutture dati annoveriamo le liste concatenate, le code, le pile (o stack) e gli alberi. In questo capitolo introdurremo i concetti fondamentali relativi ai puntatori; cercheremo di sottolineare inoltre l'intima correlazione che esiste fra array, puntatori e stringhe, e includeremo per questo motivo una serie di esercizi sulla manipolazione di stringhe. Nel Capitolo 6 studieremo come utilizzare i puntatori con le strutture. Nei Capitoli 9 e 10 vedremo come la programmazione a oggetti fa uso di puntatori e riferimenti. Nel Capitolo 4 del volume Tecniche Avanzate introdurremo le tecniche per la gestione dinamica della memoria e vedremo, per mezzo di alcuni esempi, come creare e utilizzare le strutture dati dinamiche. Il fatto di assimilare gli array e le stringhe ai puntatori è, ad ogni modo, un'eredità del C. Nei prossimi capitoli muteremo la nostra visione degli array e delle stringhe, considerandoli a tutti gli effetti come oggetti.

5.2 Come si dichiarano e si inizializzano i puntatori

A differenza delle variabili viste finora, che contengono un valore specifico, il valore di una variabile di tipo *puntatore* è un indirizzo di memoria. Più precisamente, un puntatore contiene l'indirizzo di un'altra variabile che contiene un valore specifico. In questo senso il

nome di una variabile riferisce *direttamente* un valore, mentre un puntatore lo riferisce *indirettamente* (Figura 5.1). L'operazione di riferire un valore tramite un puntatore prende il nome di *risoluzione del riferimento*.

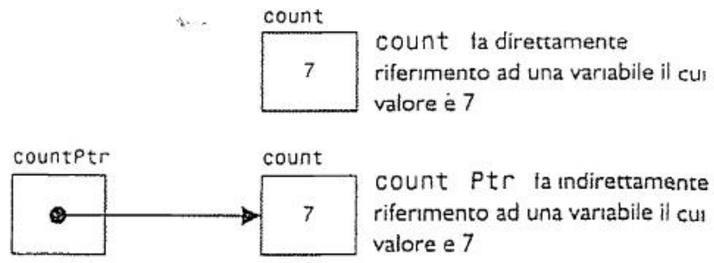


Figura 5.1 Riferimento diretto ed indiretto ad una variabile.

I puntatori, esattamente come tutte le altre variabili, devono essere dichiarati prima di poter essere utilizzati. La dichiarazione

```
int *countPtr, count;
```

dichiara la variabile `countPtr` di tipo `int *`, puntatore a un valore intero, e si legge "`countPtr` è un puntatore a un intero" La variabile `count`, dichiarata subito dopo, è un intero, non un puntatore a un intero. Il segno `*` si applica soltanto a `countPtr`. Ogni variabile puntatore si dichiara con questa notazione che fa uso dell'asterisco. Per esempio, la dichiarazione

```
float *xPtr, *yPtr;
```

indica che `xPtr` e `yPtr` sono entrambi puntatori a valori `float`. All'interno di una dichiarazione l'asterisco indica che la variabile a cui si riferisce è un puntatore. I puntatori possono puntare a oggetti di qualsiasi tipo.

Errore tipico 5.1

Se in una dichiarazione scrivete l'asterisco soltanto all'inizio, assumendo che verrà distribuito su tutte le variabili elencate dopo di esso, commettete un errore. Infatti tutte le variabili, a esclusione della prima, non saranno dichiarate come puntatori, perché ogni variabile puntatore deve essere dichiarata con il suo asterisco.

Buona abitudine 5.1

Anche se non è strettamente necessario, terminate i nomi delle variabili puntatore in `Ptr`, in modo da poter riconoscerle immediatamente e trattarle in modo corretto nel corso del programma.

I puntatori dovrebbero essere inizializzati durante la loro dichiarazione o in un assegnamento. Un puntatore può essere inizializzato a `0`, a `NULL` o a un indirizzo. Un puntatore che vale `0` o `NULL` non punta ad alcun dato. `NULL` è una costante simbolica definita nel file di intestazione `<iostream.h>` (e in molti altri file di intestazione). Inizializzare un puntatore a `NULL` o a `0` è equivalente, anche se in C++ si preferisce generalmente `0`. Quando si utilizza `0`, infatti, tale valore viene poi convertito implicitamente in un puntatore al tipo di dato corretto. Il valore `0` è l'unico che si può assegnare direttamente a un puntatore senza effettuare prima un cast. L'assegnamento di indirizzi a puntatori verrà discusso nella Sezione 5.3.

Collaudo e messa a punto 5.1

Inizializzate sempre i puntatori, in modo che non puntino ad aree di memoria non previste o non inizializzate.

5.3 Gli operatori di manipolazione dei puntatori

L'operatore `&`, è un operatore unario che restituisce l'indirizzo del suo operando, ed è detto *operatore indirizzo*. Per esempio, se scriviamo le seguenti dichiarazioni

```
int y = 5;
int *yPtr;
```

l'istruzione

```
yPtr = &y;
```

assegna l'indirizzo della variabile `y` alla variabile puntatore `yPtr`. Si dice che la variabile `yPtr` *"punta"* a `y`. La Figura 5.2 contiene una rappresentazione schematica della memoria dopo questo assegnamento. In figura rappresentiamo la relazione di "puntamento" con una freccia che parte dal puntatore e arriva all'oggetto puntato. La Figura 5.3 illustra la rappresentazione del puntatore in memoria, supponendo che la variabile intera `y` si trovi all'indirizzo di memoria `600000` e che `yPtr` si trovi all'indirizzo `500000`.

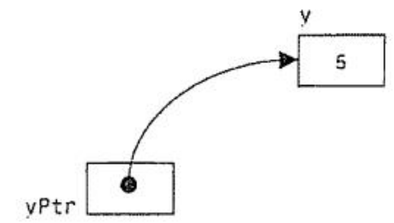


Figura 5.2 Rappresentazione grafica di un puntatore che punta ad una variabile intera.

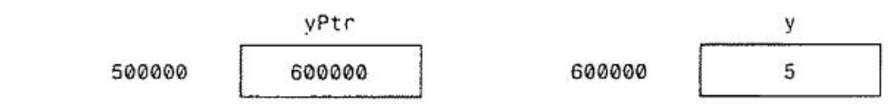


Figura 5.3 Rappresentazione di y e yPtr in memoria.

L'operando dell'operatore indirizzo deve essere un *lvalue*, ovvero un oggetto a cui si può assegnare un valore, come il nome di una variabile. L'operatore indirizzo non può essere applicato infatti a costanti, espressioni che non possono essere riferite o variabili che appartengono alla classe di memorizzazione `register`.

L'operatore `*`, chiamato comunemente *operatore di risoluzione del riferimento* o *di dereferenza* (dall'inglese *dereference*), restituisce un un *alias* dell'oggetto a cui punta il suo operando. Per esempio, rimanendo sulla Figura 5.2, l'istruzione

```
cout << *yPtr << endl;
```

visualizza il valore della variabile `y`, cioè `5`, in un modo analogo all'istruzione

```
cout << y << endl;
```

Quando si utilizza * in questo modo si dice che il puntatore *è dereferenziato* e può essere usato come la variabile a cui punta. Un puntatore dereferenziato può anche essere utilizzato a sinistra di un assegnamento, come in

```
*yPtr = 9;
```

che assegna 9 a y in Figura 5.3 oppure può essere utilizzato per leggere valori dall'input così:

```
cin >> *yPtr;
```

Un puntatore dereferenziato è un *lvalue* a tutti gli effetti.

Errore tipico 5.2

Se dereferenziate un puntatore che non è stato inizializzato correttamente, o che non punta ad alcun indirizzo di memoria specifico, si può generare un errore fatale durante l'esecuzione del programma, o potreste modificare accidentalmente dati importanti.

Errore tipico 5.3

Se dereferenziate una variabile che non è un puntatore commettete un errore di sintassi.

Errore tipico 5.4

Se dereferenziate un puntatore che vale 0 si verificherà normalmente un errore fatale in fase di esecuzione.

Il programma in Figura 5.4 mostra come si utilizzano gli operatori che abbiamo illustrato. L'operatore << visualizza le locazioni di memoria come interi esadecimali (cfr. l'appendice sui sistemi numerici per ulteriori informazioni sugli interi esadecimali).

Obiettivo portabilità 5.1

Il formato di visualizzazione di un puntatore dipende dal sistema. Alcuni sistemi visualizzano i puntatori come interi esadecimali, altri come interi decimali.

Come rileviamo dall'output, l'indirizzo di a e il valore di aPtr sono identici, e questo ci conferma che l'indirizzo di a è stato effettivamente assegnato ad aPtr. Gli operatori & e * effettuano operazioni opposte: se li applichiamo di seguito a aPtr, nell'ordine che vogliamo, il risultato visualizzato sarà lo stesso. Nella tabella di Figura 5.5 illustriamo la precedenza e l'associatività degli operatori che abbiamo incontrato finora.

```
1   // Fig. 5.4: fig05_04.cpp    Utilizzo degli operatori & e *
2   #include <iostream.h>
3
4   int main()
5   {
6      int a;         // a e un intero
7      int *aPtr;     // aPtr è un puntatore a un intero
8
9      a = 7;
10     aPtr = &a;     // in aPtr c'è ora l'indirizzo di a
11
```

Figura 5.4 Gli operatori & e * (continua)


```
12     cout << "The address of a is " << &a
13          << "\nThe value of aPtr is " << aPtr;
14
15     cout << "\n\nThe value of a is "
16          << "\nThe value of *aPtr is " << *aPtr;
17
18     cout << "\n\nShowing that * and & are inverses of "
19          << "each other.\n&*aPtr = " << &*aPtr
20          << "\n*&aPtr = " << *&aPtr << endl;
21     return 0;
22  }
```

```
The address of a is 0x0064FDF4
The value of aPtr is 0x0064FDF4
The value of a is
The value of *aPtr is 7
Showing that * and & are inverses of each other
&*aPtr = 0x0064FDF4
*&aPtr = 0x0064FDF4
```

Figura 5.4 Gli operatori & e *

Operatori	Associatività	Tipo
() []	da sinistra a destra	parentesi
++ & -- ! * + - static_cast<*type*>()	da destra a sinistra	unario
* / %	da sinistra a destra	moltiplicativo
+ -	da sinistra a destra	additivo
<< >>	da sinistra a destra	inserzione/estrazione
< <= > >=	da sinistra a destra	relazionale
== !=	da sinistra a destra	uguaglianza
&&	da sinistra a destra	AND logico
||	da sinistra a destra	OR logico
?:	da destra a sinistra	condizionale
= += -= *= /= %=	da destra a sinistra	assegnamento
,	da sinistra a destra	virgola

Figura 5.5 Precedenza e associatività degli operatori incontrati finora.

5.4 La chiamata per riferimento con argomenti di tipo puntatore

In C++ ci sono tre modi per passare un argomento a una funzione: *la chiamata per valore*, *per riferimento con argomenti di tipo riferimento* e *per riferimento con argomenti di tipo puntatore*. Nel Capitolo 3 abbiamo confrontato la chiamata per valore e quella per riferimento con argomenti di tipo riferimento, mentre in questo capitolo ci soffermiamo sulla

chiamata per riferimento con argomenti di tipo puntatore. Come abbiamo visto nel Capitolo 3, possiamo restituire un valore alla funzione chiamante dall'interno di una funzione chiamata con `return`, o possiamo utilizzare `return` semplicemente per ripassare il controllo del programma alla funzione chiamata senza passare alcun valore. Abbiamo anche visto che possiamo passare gli argomenti alle funzioni per riferimento, in modo tale che la funzione chiamata possa modificarne il valore originario o in modo da passare oggetti di grosse dimensioni a una funzione senza doverne effettuare preventivamente una copia (come accade nella chiamata per valore). Anche i puntatori, come i riferimenti, possono anche essere utilizzati per questi scopi.

È possibile utilizzare i puntatori e l'operatore di risoluzione del riferimento per simulare una chiamata per riferimento, esattamente come accadeva nei vecchi programmi C (nel quale questa era l'unica modalità consentita per effettuare la chiamata per riferimento). Se vogliamo che una funzione possa modificare i suoi argomenti, possiamo passarle gli indirizzi degli argomenti. Normalmente l'indirizzo viene passato applicando l'operatore indirizzo (`&`) alla variabile da modificare. Come abbiamo visto nel Capitolo 4, non occorre utilizzare l'operatore `&` con gli array, perché il nome di un array coincide con l'indirizzo di memoria del suo primo elemento: il nome di un array, quindi, è già un puntatore ed equivale esattamente a `&arrayName[ 0 ]`. Quando si passa l'indirizzo di una variabile a una funzione, è possibile utilizzare l'operatore di risoluzione del riferimento (`*`) per creare un alias di tale variabile: questo potrà essere utilizzato per modificare il valore presente nell'indirizzo di memoria della variabile, sempre che ciò sia possibile (cioè non sia una variabile a sola lettura).

I programmi in Figura 5.6 e 5.7 presentano due versioni di una funzione che calcola il cubo di un intero: `cubeByValue` e `cubeByReference`. In Figura 5.6 il valore della variabile viene passato a `cubeByValue` utilizzando una chiamata per valore. La funzione calcola il cubo del suo argomento e ripassa il valore alla funzione `main` con `return`. Il nuovo valore viene assegnato a `number` in `main`. Avete l'opportunità di esaminare il risultato della chiamata di funzione prima di modificare il valore della variabile. In questo programma, per esempio, avremmo anche potuto memorizzare il risultato di `cubeByValue` in un'altra variabile, esaminare il suo valore e assegnare il risultato a `number` dopo aver verificato che tale valore era accettabile.

```
1   // Fig. 5.6: fig015_06.cpp
2   // Cubo di una variabile utilizzando la chiamata per valore
3   #include <iostream.h>
4
5   int cubeByValue( int );   // prototipo
6
7   int main()
8   {
9      int number = 5;
10
11     cout << "The original value of number is " << number;
12     number = cubeByValue( number );
13     cout << "\nThe new value of number is " << number << endl;
14     return 0;
15  }
```

Figura 5.6 Calcolo del cubo di una variabile con la chiamata per valore (continua)


```
16
17  int cubeByValue( int n )
18  {
19     return n * n * n;   // effettua il cubo della variabile
20                         // locale n
21  }
```

```
The original value of number is 5
The new value of number is 125
```

Figura 5.6 Calcolo del cubo di una variabile con la chiamata per valore.

Il programma in Figura 5.7 passa la variabile `number` utilizzando la chiamata per riferimento alla funzione `cubeByReference`: ciò che viene passato è in realtà l'indirizzo di `number`. La funzione `cubeByReference` prende come argomento un puntatore a un numero intero `nPtr`. Essa dereferenzia poi il puntatore per calcolare il cubo del valore a cui punta `nPtr`. Ciò modifica il valore della variabile `number` di `main`. Notate che nell'istruzione `*nPtr = *nPtr * *nPtr * *nPtr` si sfrutta la precedenza dell'operatore di risoluzione del riferimento su quello di moltiplicazione; per rendere il codice più chiaro si sarebbe potuto scrivere `*nPtr = (*nPtr) * (*nPtr) * (*nPtr)`. Le Figura 5.8 e 5.9 analizzano graficamente i programmi in Figura 5.6 e 5.7 rispettivamente.

```
1   // Fig. 5.7: fig05_07.cpp
2   // Cubo di una variabile utilizzando la chiamata
3   // per riferimento con un argomento puntatore
4   #include <iostream.h>
5
6   void cubeByReference( int * );   // prototipo
7
8   int main()
9   {
10     int number = 5;
11
12     cout << "The original value of number is " << number;
13     cubeByReference( &number );
14     cout <<"\nThe new value of number is "<< number << endl;
15     return 0;
16  }
17
18  void cubeByReference( int *nPtr )
19  {
20     *nPtr = *nPtr * *nPtr * *nPtr;  // effettua il cubo del
21                                     // numero in main
22  }
```

```
The original value of number is 5
The new value of number is 125
```

Figura 5.7 Calcolo del cubo di una variabile con un argomento puntatore.

Prima che `main` chiami `cubeByValue`

```
int main()                                  number
{
   int number = 5;                           5

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}                                            n
                                          indefinito
```

Dopo che `cubeByValue` ha ricevuto la chiamata

```
int main()                                  number
{
   int number = 5;                           5

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}                                            n
                                             5
```

Dopo che `cubeByValue` ha fatto il cubo del parametro `n`

```
int main()                                  number
{
   int number = 5;                           5

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{                          125
   return (n * n * n;)                       n
}                                         indefinito
```

Dopo che `cubeByValue` ha ritornato il valore a `main`

```
int main()                                  number
{
   int number = 5;                           5

   number = (cubeByValue( number );)
}                          125
```

```
int cubeByValue( int n )
{
   return n * n * n;
}                                            n
                                          indefinito
```

Dopo che `main` ha completato l'assegnamento a `number`:

```
int main()                                  number
{
   int number = 5;                           125

   number = cubeByValue( number );
}
```

```
int cubeByValue( int n )
{
   return n * n * n;
}                                            n
                                          indefinito
```

Figura 5.8 Analisi di una tipica chiamata per valore.

Prima della chiamata per riferimento a `cubeByReference`:

```
int main()                                  number
{
   int number = 5;                           5

   cubeByReference( &number );
}
```

```
void cubeByreference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}                                           nPtr
                                          indefinito
```

Dopo la chiamata per riferimento a `cubeByReference` e prima di fare il cubo di `*nPtr`:

```
int main()                                  number
{
   int number = 5;                           5

   cubeByReference( &number );
}
```

```
void cubeByreference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}                                           nPtr
                                             •
```

Dopo che è stato fatto il cubo di `*nPtr`:

```
int main()                                  number
{
   int number = 5;                           5

   cubeByReference( &number );
}
```

```
void cubeByreference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}                                           nPtr
                                             •
```

Figura 5.9 Analisi di una tipica chiamata per riferimento con argomento puntatore.

Errore tipico 5.5

Se volete ottenere il valore a cui punta un puntatore, ma utilizzate quest'ultimo senza dereferenziarlo, commettete un errore.

Una funzione che riceve come argomento un indirizzo deve definire il suo parametro come puntatore. Per esempio, l'intestazione della funzione `cubeByReference` è

```
void cubeByReference( int *nPtr )
```

L'intestazione specifica che la funzione riceve come argomento l'indirizzo di una variabile intera, che memorizza localmente in `nPtr`, e che non restituisce alcun valore. Il prototipo di funzione di `cubeByReference` contiene tra parentesi la notazione `int *`. Come per gli altri, per documentare meglio i vostri programmi, il compilatore li ignorerà.

La notazione precedente, che indica un argomento puntatore, si può adottare per una funzione che riceve come argomento un array unidimensionale. Il compilatore non fa differenza tra una funzione che riceve un array unidimensionale e una funzione che riceve un puntatore. Ciò significa, tuttavia, che la funzione deve "sapere" se riceve un array o semplicemente una variabile con una chiamata per riferimento. Quando il compilatore legge un parametro array nella forma `int b[]`, lo converte automaticamente in `int *const b`, cioè in un puntatore costante a un intero. Le due forme possono essere utilizzate in modo del tutto equivalente.

Buona abitudine 5.2

Come regola, passate i parametri alle funzioni con chiamata per valore, a meno che la funzione chiamante non richieda in modo esplicito che una variabile sia modificata. Questa è un altro istanza del principio del minimo privilegio.

5.5 Privilegi di accesso e passaggio dei parametri

Il qualificatore `const` informa il compilatore che il valore di una variabile non può essere modificato.

Ingegneria del software 5.1

Utilizzate il qualificatore `const` *per realizzare il principio del minimo privilegio. Questo principio vi aiuterà a realizzare i vostri programmi con cura, riducendo il tempo necessario per il debugging e per l'eliminazione di eventuali effetti collaterali.*

Obiettivo portabilità 5.2

Sebbene `const` *sia definito negli standard di C e C++, esso non è supportato da tutti i compilatori.*

Possiamo riassumere tutte le combinazioni di passaggio dei parametri in sei casi, scegliendo se utilizzare o meno `const` con la chiamata per valore o con le due versioni di chiamata per riferimento. La scelta della combinazione più adatta per la propria applicazione dovrebbe essere guidata dal principio del minimo privilegio, in modo da consentire alle funzioni di accedere ai dati in modo sufficiente per eseguire il proprio lavoro, ma nulla di più.

Nel Capitolo 3 abbiamo visto che, in una chiamata per valore, viene prima effettuata una copia dell'argomento da passare alla funzione e, anche se la funzione modifica la copia che ha ricevuto, ciò non ha alcun effetto sul valore dell'oggetto originario. In molti casi la funzione chiamata ha bisogno di modificare il valore ricevuto per svolgere il suo compito ma ci sono casi in cui il valore non dovrebbe essere alterato dalla funzione chiamata, nemmeno se le viene passata soltanto una copia.

Prendiamo in considerazione una funzione che visualizza un array unidimensionale, e che riceve due argomenti, cioè l'array e la sua dimensione. La funzione contiene un ciclo che visualizza gli elementi dell'array uno alla volta. La dimensione dell'array serve a stabilire l'indice dell'ultimo elemento e non cambia nel corso della funzione.

Ingegneria del software 5.2

Se un valore non cambia, o non deve cambiare, nel corpo della funzione a cui viene passato, è preferibile dichiararlo come `const`*: in questo modo ci si assicura che non sia modificato accidentalmente.*

Se cercate di modificare un valore `const`, il compilatore lo rileverà e ve lo segnalerà con un avvertimento o con un messaggio di errore.

Buona abitudine 5.3

Prima di utilizzare una funzione date un'occhiata al suo prototipo per determinare i parametri che le è consentito modificare.

Ingegneria del software 5.3

Se una funzione deve modificare un solo valore, potete passarle tutti gli argomenti per valore. Infatti il valore modificato può essere restituito alla funzione chiamante con l'istruzione `return`*. Se, al contrario, una funzione deve modificare diversi valori, utilizzate la chiamata per riferimento per tutti i parametri da modificare.*

Ci sono quattro modi per passare un puntatore come parametro di una funzione: un puntatore non costante a dati non costanti, un puntatore non costante a dati costanti, un puntatore costante a dati non costanti e un puntatore costante a dati costanti. Ogni combinazione comporta privilegi d'accesso diversi. L'accesso più ampio è senz'altro quello del primo modo, puntatore non costante a dati non costanti: in questo caso i dati possono essere modificati dereferenziando il puntatore, e anche quest ultimo può essere modificato, in modo che punti ad altri dati. Nella dichiarazione di questo tipo di puntatore non si utilizza il qualificatore `const`. Ad esempio, si può utilizzare un puntatore di questo genere se si vuole passare una stringa a una funzione che modifica tutti i suoi caratteri accedendo a ognuno di essi utilizzando l'aritmetica dei puntatori. La funzione `convertToUppercase` in Figura 5.10 dichiara il parametro `sPtr` (`char *sPtr`) come puntatore non costante a dati non costanti. La funzione accede alla stringa `s` un carattere alla volta con l'aritmetica dei puntatori. I caratteri compresi tra `'a'` e `'z'` sono sostituiti con i rispettivi caratteri maiuscoli dalla funzione `toupper`, mentre gli altri restano inalterati. La funzione riceve un solo argomento: se il carattere è una lettera minuscola, restituisce la corrispondente lettera maiuscola, altrimenti restituisce il carattere ricevuto. La funzione `toupper` fa parte delle funzioni di libreria che operano sui caratteri (occorre includere il file di intestazione `ctype.h`).

Un puntatore non costante a dati costanti è un puntatore che può essere modificato per puntare a qualsiasi dato del tipo appropriato, ma quest'ultimo non può essere modificato dereferenziando il puntatore. Un puntatore di questo genere può servire a passare un array a una funzione che deve poter accedere ai singoli elementi ma non modificarli. Per esempio la funzione `printCharacters` in Figura 5.11 dichiara il parametro `sPtr` come un puntatore di tipo `const char *` Una dichiarazione di questo tipo si legge: "`sPtr` è un puntatore a un carattere costante" Il corpo della funzione contiene un costrutto `for` che visualizza la stringa un carattere alla volta finché non incontra il carattere nullo. Dopo aver visualizzato un carattere, la funzione incrementa il puntatore `sPtr` in modo che punti al carattere successivo della stringa.

```
1  // Fig. 5.10: fig05_10.cpp  Conversione di minuscole in maiusco-
2  // le utilizzando un puntatore non costante a dati non costanti
3  #include <iostream.h>
4  #include <ctype.h>
5
6  void convertToUppercase( char * );
7
8  int main()
9  {
10    char string[] = "characters and $32.98";
11
12    cout << "The string before conversion is: " << string;
13    convertToUppercase( string );
```

Figura 5.10 Conversione di una stringa nella sua versione maiuscola (continua)

```
14       cout << "\nThe string after conversion is:
15            << string << endl;
16       return 0;
17    }
18
19    void convertToUppercase( char *sPtr )
20    {
21       while ( *sPtr != '\0' ) {
22
23          if ( *sPtr >= 'a' && *sPtr <= 'z' )
24             *sPtr=toupper(*sPtr);  // conversione in maiuscolo
25
26          ++sPtr;  // sposta sPtr sul carattere successivo
27       }
28    }
```

```
The string before conversion is: characters and $32.98
The string after conversion is:  CHARACTERS AND $32.98
```

Figura 5.10 Conversione di una stringa nella sua versione maiuscola.

```
1    // Fig. 5.11: fig05_11.cpp
2    // Visualizzazione di una stringa un carattere alla volta
3    // utilizzando un puntatore non costante a dati costanti
4    #include <iostream.h>
5
6    void printCharacters( const char * );
7
8    int main()
9    {
10      char string[] = "print characters of a string";
11      cout << "The string is:\n";
12      printCharacters( string );
13      cout << endl;
14      return 0;
15   }
16   // In printCharacters, sPtr è un puntatore a un carattere
17   // costante. I caratteri non possono essere modificati
18   // tramite sPtr (cioè sPtr è un puntatore "a sola lettura")
19   void printCharacters( const char *sPtr )
20   {
21      for( ; *sPtr != '\0'; sPtr++) // nessuna inizializzazione
22         cout << *sPtr;
23   }
```

```
The string is:
print characters of a string
```

Figura 5.11 Visualizzazione di una stringa un carattere alla volta tramite un puntatore non costante a dati costanti.

La Figura 5.12 mostra gli errori di sintassi generati dal compilatore se una funzione riceve un puntatore non costante a dati costanti, ma lo utilizza poi per modificare i dati.

```
1    // Fig. 5.12: fig05_12.cpp
2    // Tentativo di modificare i dati attraverso
3    // un puntatore non costante a dati costanti.
4    #include <iostream.h>
5
6    void f( const int * );
7
8    int main()
9    {
10      int y;
11
12      f( &y );     // f tenta una modifica non permessa
13
14      return 0;
15   }
16
17   // In f, xPtr è un puntatore a un intero costante
18   void f( const int *xPtr )
19   {
20      *xPtr = 100;   // non è possibile modificare un oggetto const
21   }
```

```
Compiling FIG05_12.CPP:
Error FIG05_12.CPP 20: Cannot modify a const object
Warning FIG05_12.CPP 21: Parameter 'xPtr' is never used
```

Figura 5.12 Tentativo di modificare i dati tramite un puntatore non costante a dati costanti.

Come sappiamo, gli array sono tipi aggregati che contengono elementi della stessa natura. Nel Capitolo 6 parleremo di un altro tipo di dato aggregato detto *struttura* o, in altri linguaggi, *record*. Una struttura è in grado di contenere elementi correlati ma che sono di tipi diversi: ciò è utile, per esempio, nel caso delle informazioni degli impiegati di un azienda. Ricordiamo che il passaggio degli array come argomenti avviene simulando una chiamata per riferimento; le strutture, invece, sono passate per valore, ovvero viene passata soltanto una copia della struttura originaria. Una chiamata per valore, dunque, richiede un certo tempo di elaborazione: è necessario, infatti, effettuare la copia della struttura e poi memorizzarla nello stack, cioè nella zona di memoria in cui sono allocate le variabili locali di una funzione. Se vogliamo passare i dati di una struttura a una funzione, possiamo utilizzare un puntatore (o un riferimento) a dati costanti, recuperando l'efficienza di una chiamata per riferimento e conservando la sicurezza di una chiamata per valore. Se passiamo un puntatore a una struttura, verrà effettuata solo una copia dell'indirizzo di tale struttura. Sulle macchine che codificano gli indirizzi di memoria in 4 byte ciò significa che viene fatta la copia di un dato di soli 4 byte, anziché la copia di centinaia o anche migliaia di byte, che sono le dimensioni tipiche di una struttura.

Obiettivo efficienza 5.1

Come regola, passate dati di grosse dimensioni, come le strutture, con puntatori o riferimenti a dati costanti. In questo modo recupererete l'efficienza di una chiamata per riferimento conservando la sicurezza tipica di una chiamata per valore.

Un puntatore costante a dati non costanti punta sempre allo stesso indirizzo di memoria, ma i dati a cui punta possono essere modificati attraverso di esso. Questa è la situazione standard per il nome di un array: esso è un puntatore costante all'indirizzo di partenza dell'array. L'accesso e la modifica di tutti i dati dell'array sono possibili utilizzando il nome dell'array e gli indici. Un puntatore costante a dati non costanti può essere utilizzato, ad esempio, per passare un array a una funzione che accede ai singoli elementi unicamente tramite gli indici. I puntatori `const` devono essere inizializzati durante la loro dichiarazione: se un puntatore è parametro di una funzione, viene inizializzato con il puntatore passato alla funzione. Il programma in Figura 5.13 tenta di modificare un puntatore costante. Il puntatore `ptr` è infatti di tipo `int * const`. La dichiarazione si legge "`ptr` è un puntatore costante a un intero". Il puntatore viene inizializzato con l'indirizzo della variabile intera `x`. Il programma tenta dunque di assegnare l'indirizzo di `y` a `ptr`, ma il compilatore non lo permette e genera un messaggio di errore. Osservate, invece, che non viene generato alcun messaggio di errore quando si assegna a `*ptr` il valore 7, perché il valore a cui punta `ptr` può essere modificato.

```
1   // Fig. 5.13: fig05_13.cpp
2   // Tentativo di modificare un puntatore costante a
3   // dati non costanti
4   #include <iostream.h>
5
6   int main()
7   {
8      int x, y;
9
10     int * const ptr = &x; //
11     // ptr è un puntatore costante a un intero. Si puo
12     // modificare l'intero tramite ptr, ma ptr deve sempre
13     // puntare allo stesso indirizzo di memoria.
14     *ptr = 7;
15     ptr = &y;
16
17     return 0;
18  }
```

```
Compiling FIG05_13.CPP:
Error FIG05_13.CPP 15: Cannot modify a const object
Warning FIG05_13.CPP 18: 'y' is declared but never used
```

Figura 5.13 Tentativo di modificare un puntatore costante a dati non costanti.

Errore tipico 5.6

Se non inizializzate un puntatore `const` *commettete un errore di sintassi.*

Il privilegio d'accesso minimo in assoluto è quello di un puntatore costante a dati costanti: un puntatore di questo genere punta sempre allo stesso indirizzo di memoria e i dati a cui punta non possono essere modificati. Questo è il modo in cui conviene passare un array a una funzione che deve soltanto leggerlo, ma non modificarlo. Il programma in Figura 5.14 dichiara la variabile puntatore `ptr` di tipo `int * const`. Questa dichiarazione si legge "`ptr` è un puntatore costante a un intero costante"

La figura mostra i messaggi di errore che genera il compilatore quando il programma tenta di modificare i dati a cui punta `ptr`, e quando tenta di modificare l'indirizzo stesso di `ptr`. Come vedete, invece, non viene generato alcun errore quando il programma cerca di visualizzare il valore a cui punta `ptr`, perché tale operazione di output non comporta alcuna modifica sui dati.

```
1   // Fig. 5.14: fig05_14.cpp
2   // Tentativo di modificare un puntatore costante a
3   // dati costanti.
4   #include <iostream.h>
5
6   int main()
7   {
8      int x = 5, y;
9
10     const int *const ptr = &x;
11       // ptr è un puntatore costante a unintero costante.
12       // ptr punta sempre allo stesso indirizzo di memoria
13       // e l'intero a cui punta non può essere modificato.
14
15     cout << *ptr << endl;
16     *ptr = 7;
17     ptr = &y;
18
19     return 0;
20  }
```

```
Compiling FIG05_14.CPP:
Error FIG05_14.CPP 16: Cannot modify a const object
Error FIG05_14.CPP 17: Cannot modify a const object
Warning FIG05_14.CPP 20: 'y' is declared but never used
```

Figura 5.14 Tentativo di modificare un puntatore costante a dati costanti.

5.6 L'algoritmo di ordinamento a bolle con i puntatori

Torniamo sul programma *bubble sort* in Figura 4.16 e riscriviamolo inserendo due nuove funzioni: `bubbleSort` e `swap` (Figura 5.15). La funzione `bubbleSort` ordina l'array: essa chiama la funzione `swap` per scambiare di posto gli elementi `array[ j ]` e `array[ j + 1 ]`. Ricordate che il C++ si basa sul principio di occultamento delle informazioni tra le funzioni, per cui `swap` non può accedere ai singoli elementi dell'array di `bubbleSort`. Dal

momento che **bubbleSort** richiede espressamente che **swap** acceda ai singoli elementi dell'array da scambiare, **bubbleSort** passa a **swap** tali elementi per riferimento, cioè passa in realtà l'indirizzo degli elementi. Anche se gli array sono passati automaticamente con chiamata per riferimento, i singoli elementi di un array sono dei valori scalari e quindi sono passati normalmente per valore. Per questo motivo **bubbleSort** deve utilizzare l'operatore indirizzo (**&**) su ciascun elemento che vuole passare a **swap**, in questo modo:

```
swap( &array[ j ], &array[ j + 1 ]);
```

La funzione **swap** riceve **&array[j]** nella variabile puntatore **element1Ptr**; per il principio di occultamento delle informazioni, **swap** non può venire a conoscenza del nome **array[j]**, ma può utilizzare ***element1Ptr** come alias di **array[j]**. Quando **swap** riferisce ***element1Ptr**, in realtà è come se riferisse **array[j]** di **bubbleSort**. In *modo simile quando swap riferisce* ***element2Ptr****, in realtà è come se riferisse* **array[j + 1]** in **bubbleSort**. Anche se in **swap** non è consentito scrivere

```
hold = array[ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = hold;
```

si ottiene lo stesso effetto con queste istruzioni

```
hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

```
1  // Fig. 5.15: fig05_15.cpp
2  // Questo programma mette dei valori in un array, li pone
3  // in ordine ascendente e visualizza l'array risultante.
4  #include <iostream.h>
5  #include <iomanip.h>
6
7  void bubbleSort( int *, const int );
8
9  int main()
10 {
11    const int arraySize = 10;
12    int a[arraySize] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
13    int i;
14
15    cout << "Data items in original order\n";
16
17    for ( i = 0; i < arraySize; i++ )
18       cout << setw( 4 ) << a[ i ];
19
20    bubbleSort( a, arraySize );          // ordinamento
21    cout << "\nData items in ascending order\n";
22
23    for ( i = 0; i < arraySize; i++ )
24       cout << setw( 4 ) << a[ i ];
25
```

Figura 5.15 Bubble sort con chiamata per riferimento (continua)


```
26    cout << endl;
27    return 0;
28 }
29
30 void bubbleSort( int *array, const int size )
31 {
32    void swap( int *, int * );
33
34    for ( int pass = 0; pass < size - 1; pass++ )
35
36       for ( int j = 0; j < size - 1; j++ )
37
38          if ( array[ j ] > array[ j + 1 ] )
39             swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44    int hold = *element1Ptr;
45    *element1Ptr = *element2Ptr;
46    *element2Ptr = hold;
47 }
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37

Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Figura 5.15 Bubble sort con chiamata per riferimento.

Analizziamo ora alcune caratteristiche di **bubbleSort**: la sua intestazione dichiara l'array come **int *array** anziché come **int array[]**, indicando di ricevere come argomento un array unidimensionale (ricordate che le notazioni sono equivalenti). Il parametro **size** è dichiarato come **const** per applicare il principio del minimo privilegio. La dimensione dell'array rimane immutata durante l'esecuzione di **bubbleSort**, di conseguenza **size** viene dichiarato **const**, per assicurare che non possa essere modificato. Se la dimensione dell'array venisse modificata durante l'ordinamento, l'algoritmo non funzionerebbe correttamente.

Il prototipo di **swap** è incluso nel corpo della funzione **bubbleSort** perché essa è l'unica funzione a chiamare **swap**. Scrivendo il prototipo in **bubbleSort** si impone una restrizione sulle funzioni che possono chiamare **swap**: le uniche chiamate possibili sono quelle dall'interno di **bubbleSort**. Le altre funzioni non potrebbero chiamare **swap**, perché non possono accedere al suo prototipo e, come sapete, in C++ i prototipi di funzione sono obbligatori.

Ingegneria del software 5.4

Scrivere il prototipo di una funzione all'interno della definizione di altre funzioni costituisce un'applicazione del principio del minimo privilegio. Infatti le uniche funzioni che possono richiamare tale funzione sono quelle che contengono il suo prototipo.

Osservate che la funzione `bubbleSort` riceve la dimensione dell'array come parametro. La funzione deve necessariamente conoscere l'estensione dell'array per poterlo ordinare, ma quando si passa un array a una funzione, questa riceve in realtà solo l'indirizzo di memoria iniziale dell'array.

Includendo il parametro della dimensione dell'array nella funzione `bubbleSort` l'abbiamo generalizzata: ora è possibile riutilizzarla in qualsiasi altro programma che ordina array di interi unidimensionali, indipendentemente dalla loro dimensione.

Ingegneria del software 5.5

Se passate un array a una funzione, ricordatevi di passarle anche la dimensione dell'array, anziché incorporare questo dato all'interno della funzione. In questo modo generalizzerete la funzione e potrete riutilizzarla in altri programmi.

Avremmo potuto includere la dimensione dell'array direttamente all'interno della funzione, ma avremmo ottenuto una funzione capace di lavorare in modo specifico su un solo tipo array, cioè su un array di quella dimensione, per cui potremmo difficilmente riutilizzarla in altri programmi.

Il C++ prevede l'operatore unario `sizeof`, che determina la dimensione in byte di un array o di qualunque tipo di dato in fase di compilazione. Se applichiamo `sizeof` all'array di Figura 5.16, otteniamo il numero totale di byte presenti nell'array nella forma di un valore `size_t`, che normalmente è sinonimo di `unsigned int`. Il computer che abbiamo utilizzato per i nostri esempi memorizza le variabili di tipo `float` in 4 byte di memoria, per cui un array di 20 elementi `float` occupa 80 byte. Ma attenzione, se applichiamo `sizeof` a un parametro puntatore in una funzione che riceve un array come argomento, otteniamo la dimensione del puntatore in byte (4) e non la dimensione dell'array.

Errore tipico 5.7

Se utilizzate l'operatore `sizeof` in una funzione per determinare la dimensione in byte di un array, ottenete la dimensione in byte di un puntatore anziché dell'array.

```
1   // Fig. 5.16: fig05_16.cpp
2   // L'operatore sizeof utilizzato sul nome di un array
3   // restituisce il numero di byte occupati dall'array.
4   #include <iostream.h>
5
6   size_t getSize( float * );
7
8   int main()
9   {
10     float array[ 20 ];
11
```

Figura 5.16 L'operatore sizeof applicato al nome di un array restituisce il numero di byte occupati dall'array (continua)


```
12     cout << "The number of bytes in the array is "
13          << sizeof( array )
14          << "\nThe number of bytes returned by getSize is "
15          << getSize( array ) << endl;
16
17     return 0;
18  }
19
20  size_t getSize( float *ptr )
21  {
22     return sizeof( ptr );
23  }
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

Figura 5.16 L'operatore sizeof applicato al nome di un array restituisce il numero di byte occupati dall'array.

Il numero di elementi di un array può anche essere determinato dal risultato di due operazioni `sizeof`. Per esempio, osservate la seguente dichiarazione di array:

```
double realArray[ 22 ];
```

Se le variabili di tipo `double` occupano 8 byte di memoria, l'array `realArray` ne occupa in tutto 176. Per determinare il numero di elementi dell'array è possibile utilizzare questa espressione:

```
sizeof realArray / sizeof( double )
```

In questa espressione determiniamo il numero di byte di `realArray` e lo dividiamo per il numero di byte che occupa un solo valore `double`.

Obiettivo portabilità 5.3

Il numero di byte occupati da un tipo di dato varia da sistema a sistema. Se scrivete programmi che dipendono dalle dimensioni dei tipi e prevedete di eseguirli su sistemi diversi, utilizzate `sizeof` per determinare il numero di byte occupati dai diversi tipi di dato.

Il programma in Figura 5.17 fa uso di `sizeof` per calcolare il numero di byte occupati dai tipi standard sul calcolatore da noi utilizzato:

```
1   // Fig. 5.17: fig05_17.cpp
2   // Esempio di utilizzo dell'operatore sizeof.
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      char c;
9      short s;
```

Figura 5.17 L'operatore sizeof determina le dimensioni dei tipi di dato standard (continua)

```
10      int i;
11      long l;
12      float f;
13      double d;
14      long double ld;
15      int array[ 20 ], *ptr = array;
16
17      cout << "sizeof c = " << sizeof c
18           << "\tsizeof(char) = " << sizeof( char )
19           << "\nsizeof s = " << sizeof s
20           << "\tsizeof(short) = " << sizeof( short )
21           << "\nsizeof i = " << sizeof i
22           << "\tsizeof(int) = " << sizeof( int )
23           << "\nsizeof l = " << sizeof l
24           << "\tsizeof(long) = " << sizeof( long )
25           << "\nsizeof f = " << sizeof f
26           << "\tsizeof(float) = " << sizeof( float )
27           << "\nsizeof d = " << sizeof d
28           << "\tsizeof(double) = " << sizeof( double )
29           << "\nsizeof ld = " << sizeof ld
30           << "\tsizeof(long double)=" << sizeof(long double)
31           << "\nsizeof array = " << sizeof array
32           << "\nsizeof ptr = " << sizeof ptr
33           << endl;
34      return 0;
35  }
```

```
sizeof c = 1     sizeof(char) = 1
sizeof s = 2     sizeof(short) = 2
sizeof i = 4     sizeof(int) = 4
sizeof l = 4     sizeof(long) = 4
sizeof f = 4     sizeof(float) = 4
sizeof d = 8     sizeof(double) = 8
sizeof ld = 8    sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Figura 5.17 L'operatore sizeof determina le dimensioni dei tipi di dato standard.

L'operatore `sizeof` si può applicare a un nome di variabile, a un nome di tipo o a un valore costante. Se si applica a un nome di variabile (che non sia un array) o a un valore costante, restituisce il numero di byte occupato da quella variabile o costante specifica. Ricordate che le parentesi tonde dopo `sizeof` sono obbligatorie se l'operando è il nome di un tipo di dato, mentre non lo sono se è il nome di una variabile. Ricordare, infine, che `sizeof` è un operatore e non una funzione.

Errore tipico 5.8

Se omettete le parentesi in un'operazione `sizeof` *e l'operando è il nome di un tipo di dato, commettete un errore di sintassi.*

Obiettivo efficienza 5.2

`sizeof` *è un operatore unario che viene applicato durante la compilazione, e non durante l'esecuzione di una funzione. Perciò* `sizeof` *non ha alcun impatto sull'efficienza di una funzione.*

5.7 L'aritmetica dei puntatori

I puntatori possono essere utilizzati come operandi in espressioni aritmetiche, di assegnamento e di confronto, anche se non tutti gli operatori di queste espressioni possono agire su di essi. In questa sezione descriveremo gli operatori che possono agire sui puntatori.

Sui puntatori è possibile effettuare un insieme limitato di operazioni aritmetiche: un puntatore può essere incrementato (++) o decrementato (--), è possibile aggiungergli o sottrargli un numero intero (+ o +=, - o -=) e infine un puntatore può essere sottratto da un altro puntatore.

Supponiamo che il compilatore abbia posto l'array `int v[ 5 ]` all'indirizzo di memoria **3000**. Se il puntatore `vPtr` è stato inizializzato per puntare a `v[ 0 ]`, il valore di `vPtr` è **3000**. In Figura 5.18 abbiamo descritto graficamente questa situazione, nell'ipotesi che *la macchina rappresenti gli interi utilizzando 4 byte. Notate che* `vPtr` *può essere inizializzato* per puntare all'array v con una delle due istruzioni che seguono:

```
vPtr = v;
vPtr = &v[ 0 ];
```

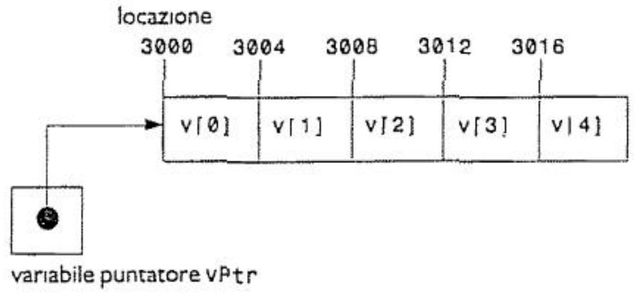


Figura 5.18 L'array v e la variabile puntatore vPtr che punta a v.

Obiettivo portabilità 5.4

La maggior parte dei computer moderni ha interi a 2 o 4 byte. Le macchine più recenti utilizzano interi anche di 8 byte. Dal momento che l'aritmetica dei puntatori dipende dalla dimensione dell'oggetto a cui si punta, tale aritmetica dipende dalla macchina.

Nell'aritmetica convenzionale l'addizione `3000 + 2` dà come risultato **3002**. La stessa semplicità non vale in genere per l'aritmetica dei puntatori: se si aggiunge o si sottrae un valore intero da un puntatore, esso non viene soltanto incrementato o decrementato di quel valore, ma di quel valore moltiplicato per la dimensione dell'oggetto a cui punta. Per esempio, l'istruzione

```
vPtr += 2;
```

darebbe come risultato 3008, cioè 3000 + 2 * 4, se un intero occupa 4 byte di memoria. Nell'array v, vPtr punterà ora a v[2] (Figura 5.19). Se un intero occupa soltanto 2 byte di memoria, il calcolo precedente darà come risultato l'indirizzo di memoria 3004, cioè 3000 + 2 * 2. Se l'array fosse di dimensioni diverse, l'istruzione precedente incrementerebbe il puntatore per il doppio del numero di byte occupati da un oggetto di quel tipo di dato. Se applicate l'aritmetica dei puntatori a un array di caratteri, i risultati saranno uguali a quelli dell'aritmetica convenzionale, perché un carattere occupa un solo byte di memoria.

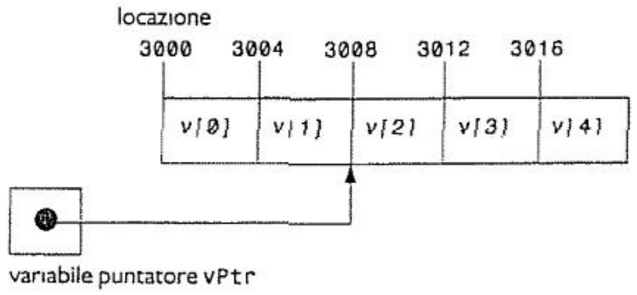


Figura 5.19 Il puntatore vPtr dopo le operazioni di aritmetica dei puntatori.

Se vPtr è stato incrementato a 3016, cioè punta a v[4], l'istruzione

```
vPtr -= 4;
```

ripristinerebbe il valore originario di vPtr, ovvero 3000, vale a dire l'indirizzo iniziale dell'array. Se un puntatore deve essere incrementato o decrementato di uno, si possono utilizzare gli operatori di incremento (++) e decremento (--). Le due istruzioni

```
++vPtr;
vPtr++;
```

incrementano ciascuna il puntatore, in modo che punti all'elemento successivo dell'array. Le due istruzioni

```
--vPtr;
vPtr--;
```

decrementano ciascuna il puntatore, in modo che punti all'elemento precedente dell'array.

Le variabili puntatore possono essere sottratte l'una dall'altra. Per esempio, se vPtr contiene l'indirizzo 3000 e v2Ptr l'indirizzo 3008, l'istruzione

```
x = v2Ptr - vPtr;
```

assegna a x il numero di elementi dell'array compresi tra vPtr a v2Ptr, in questo caso 2. Non ha senso utilizzare l'aritmetica dei puntatori in casi diversi dagli array: infatti, non possiamo supporre che due variabili dello stesso tipo si trovino in locazioni di memoria contigue, a meno che non si tratti degli elementi di uno stesso array.

Errore tipico 5.9

Se applicate l'aritmetica dei puntatori a un puntatore che non si riferisce ad un array, è probabile che stiate commettendo un errore logico.

Errore tipico 5.10

Se sottraete o confrontate due puntatori che non riferiscono elementi dello stesso array, è probabile che stiate commettendo un errore logico.

Errore tipico 5.11

Se applicando l'aritmetica dei puntatori oltrepassate i limiti dell'array, in uno dei due sensi, è probabile che stiate commettendo un errore logico.

Un puntatore può essere assegnato a un altro puntatore, se entrambi sono dello stesso tipo. In caso contrario, si deve utilizzare l'operatore cast per convertire il valore del puntatore a destra del segno di assegnamento nel tipo di puntatore a sinistra. L'unica eccezione è quella dei puntatori al tipo void (void *), che sono puntatori generici e possono rappresentare qualsiasi tipo di puntatore. Per questo motivo tutti i tipi di puntatore possono essere assegnati a un puntatore void senza effettuarne preventivamente un cast. Vogliamo sottolineare, però, che non vale il contrario, perché un puntatore void non può essere assegnato a un altro tipo di puntatore senza prima effettuarne un cast.

Un puntatore void * non può essere dereferenziato e il motivo è semplice: il compilatore sa, per esempio, che il puntatore a un int fa riferimento a quattro byte di memoria (su una macchina con interi di 4 byte), ma un puntatore a un tipo void contiene un indirizzo di memoria per un tipo di dato non specificato. Il compilatore non sa a quanti byte fa riferimento tale puntatore e, in generale, ha bisogno di conoscere il tipo di dato puntato per determinare il numero di byte da dereferenziare. Nel caso di un puntatore al tipo void, questo numero non è definito.

Errore tipico 5.12

*Se assegnate un puntatore di un dato tipo a un puntatore di tipo diverso (che non sia void *) senza effettuare il cast sul primo puntatore, commettete un errore di sintassi.*

Errore tipico 5.13

*Se dereferenziate un puntatore void *, commettete un errore di sintassi.*

È possibile confrontare due puntatori utilizzando gli operatori relazionali e di uguaglianza, ma, anche in questo caso, queste operazioni hanno senso solo se i puntatori puntano a elementi di uno stesso array. I confronti tra puntatori si traducono in confronti su indirizzi. Il confronto tra due puntatori che puntano allo stesso array può indicare, per esempio, se un puntatore punta a un elemento di indice più alto rispetto all'altro. Gli operatori di uguaglianza si utilizzano spesso per determinare se un puntatore è uguale a 0.

5.8 La correlazione tra puntatori e array

Array e puntatori sono strettamente correlati in C++ e possono essere utilizzati in modo equivalente. Il nome di un array è in realtà un puntatore costante. I puntatori, poi, possono essere utilizzati in tutte le operazioni di indicizzazione di un array.

Buona abitudine 5.4

Nelle manipolazioni di un array utilizzate la notazione degli array piuttosto che quella dei puntatori. Anche se il programma sarà compilato un po' più lentamente, quasi certamente risulterà più facile da leggere.

Supponiamo di dichiarare l'array di interi b[5] e la variabile puntatore a un intero bptr. Il nome dell'array b senza indici è un puntatore al primo elemento dell'array, per cui possiamo inizializzare bPtr con l'indirizzo del primo elemento di b con l'istruzione

```
bPtr = b;
```

Ciò equivale a ricavare l'indirizzo del primo elemento dell'array, in questo modo

```
bPtr = &b[ 0 ];
```

D'ora in poi possiamo utilizzare una notazione alternativa per riferici, ad esempio, all'elemento `b[ 3 ]` dell'array, servendoci di un'espressione puntatore

```
*( bPtr + 3 )
```

Il numero `3` è detto *offset* (o *scostamento*) del puntatore. Se il puntatore punta all'indirizzo di partenza dell'array, l'offset indica quale elemento dell'array si vuole riferire: offset e indice hanno lo stesso valore. Questa notazione prende il nome di *notazione puntatore/offset*. Le parentesi sono necessarie perché la precedenza di * è più alta di quella di +. Senza le parentesi il significato dell'operazione consisterebbe nell'aggiungere il valore 3 all'espressione `*bPtr`, per cui `b[ 0 ]` risulterebbe incrementato di 3, il che non è esattamente ciò che volevamo. Un singolo elemento di un array può essere riferito tramite un puntatore, perché l'espressione

```
&b[ 3 ]
```

equivale a

```
bPtr + 3
```

Lo stesso array può essere trattato come un puntatore, e utilizzato di conseguenza nell'aritmetica dei puntatori. Per esempio, l'espressione

```
*( b + 3 )
```

riferisce l'elemento `b[ 3 ]`. In generale, tutte le espressioni di indicizzazione di un array si possono tradurre nella notazione puntatore/offset. In questo caso non utilizziamo neanche una variabile puntatore, basta il nome dello stesso array. Osservate come l'istruzione precedente non modifichi in alcun modo il nome dell'array: b punta in modo costante al primo elemento dell'array.

D'altro canto anche i puntatori, proprio come gli array, possono essere indicizzati. Per esempio, l'espressione

```
bPtr[ 1 ]
```

riferisce l'elemento dell'array `b[ 1 ]`. Questa notazione prende il nome di *notazione puntatore/indice*. Ricordate che il nome di un array è un puntatore costante, per cui non può puntare ad altro che al suo primo elemento. Perciò l'espressione

```
b += 3
```

non è valida, perché tenta di modificare il valore associato al nome dell'array.

Errore tipico 5.14

Il nome di un array è un puntatore all'indirizzo di partenza dell'array, ma si tratta di un puntatore costante, per cui non può essere modificato nelle operazioni di aritmetica dei puntatori e deve conservare il suo valore nel corso dell'intero blocco in cui esiste.

Il programma in Figura 5.20 utilizza i quattro metodi di cui abbiamo parlato per riferire gli elementi di un array: indicizzazione dell'array, notazione puntatore/offset con il nome dell'array come puntatore, indicizzazione di un puntatore e notazione puntatore/offset con un puntatore. Scopo del programma è visualizzare i quattro elementi dell'array b.


```
1  // Fig. 5.20: fig05_20.cpp  Utilizzo degli indici e della
2  // notazione con i puntatori su array
3
4  #include <iostream.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 };
9     int *bPtr = b;   // bPtr punta ora all'array b
10
11    cout << "Array b printed with:\n"
12         << "Array subscript notation\n";
13
14    for ( int i = 0; i < 4; i++ )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17
18    cout << "\nPointer/offset notation where\n"
19         << "the pointer is the array name\n";
20
21    for ( int offset = 0; offset < 4; offset++ )
22       cout << "*(b + " << offset << ") = "
23            << *( b + offset ) << '\n';
24
25
26    cout << "\nPointer subscript notation\n";
27
28    for ( i = 0; i < 4; i++ )
29       cout << "bPtr[" << i << "] = " << bPtr[ i ] << '\n';
30
31    cout << "\nPointer/offset notation\n";
32
33    for ( offset = 0; offset < 4; offset++ )
34       cout << "*(bPtr + " << offset << ") = "
35            << *( bPtr + offset ) << '\n';
36
37    return 0;
38 }
```

```
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

Figura 5.20 Utilizzo dei quattro metodi per riferire gli elementi di un array (continua)

```
Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Figura 5.20 *Utilizzo dei quattro metodi per riferire gli elementi di un array.*

Per cogliere in modo immediato l'equivalenza di array e puntatori osservate le due funzioni per la copia di stringhe (**copy1** e **copy2**) nel programma di Figura 5.21. Scopo di entrambe è copiare una stringa in un array di caratteri. Se confrontiamo i loro prototipi, le funzioni possono sembrarci identiche. Tuttavia esse sono implementate in due modi diversi.

La funzione **copy1** utilizza l'indicizzazione dell'array per copiare la stringa **s2** nell'array di caratteri **s1**. La funzione dichiara un contatore intero **i** e lo usa come indice dell'array. È l'intestazione del costrutto **for** a effettuare l'intera operazione di copia: come vedete il suo corpo è un'istruzione vuota. L'intestazione specifica che **i** viene prima azzerata e poi incrementata di uno a ogni iterazione. La condizione di **for**, **(s1[i] = s2[i]) != '\0'**, effettua anche la copia di **s2** in **s1**, un carattere alla volta. Quando incontra il carattere nullo in **s2**, lo assegna a **s1** e il ciclo termina, perché il carattere nullo soddisfa ora la condizione di uguaglianza a **'\0'**

Ricordate che il valore di un'istruzione di assegnamento coincide con il valore assegnato all'argomento di sinistra.

```
1  // Fig. 5.21: fig05_21.cpp
2  // Copia di una stringa utilizzando la notazione degli array
3  // e dei puntatori.
4  #include <iostream.h>
5
6  void copy1( char *, const char * );
7  void copy2( char *, const char * );
8
```

Figura 5.21 Copia di una stringa utilizzando la notazione degli array e quella dei puntatori (continua)


```
9  int main()
10 {
11    char string1[ 10 ], *string2 = "Hello",
12         string3[ 10 ], string4[] = "Good Bye";
13
14    copy1( string1, string2 );
15    cout << "string1 = " << string1 << endl;
16
17    copy2( string3, string4 );
18    cout << "string3 = " << string3 << endl;
19
20    return 0;
21 }
22
23 // copia s2 in s1 utilizzando la notazione degli array
24 void copy1( char *s1, const char *s2 )
25 {
26    for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
27       ;   // il corpo non effettua alcuna operazione
28 }
29
30 // copia s2 in s1 utilizzando la notazione dei puntatori
31 void copy2( char *s1, const char *s2 )
32 {
33    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
34       ;   // // il corpo non effettua alcuna operazione
35 }
```

```
string1 = Hello
string3 = Good Bye
```

Figura 5.21 Copia di una stringa utilizzando la notazione degli array e quella dei puntatori.

La funzione **copy2** utilizza l'aritmetica dei puntatori per copiare la stringa **s2** nell'array di caratteri **s1**. Anche in questo caso l'intera operazione è effettuata dall'intestazione del costrutto **for** ma questa volta non è presente alcuna inizializzazione di variabile. Come in **copy1**, è la condizione **(*s1 = *s2) != '\0'** a effettuare la copia. Il puntatore **s2** viene dereferenziato e il carattere restituito viene assegnato al puntatore dereferenziato **s1**. Dopo l'assegnamento presente all'interno della condizione, i puntatori sono incrementati di uno, e puntano all'elemento successivo della stringa e dell'array di caratteri rispettivamente. Quando viene incontrato il carattere nullo di **s2**, questo viene assegnato al puntatore dereferenziato **s1** e il ciclo termina.

Notate che il primo argomento di entrambe le funzioni **copy1** e **copy2** deve essere un array abbastanza grande per contenere la stringa ricevuta come secondo argomento. Altrimenti, se si cercasse di scrivere in un indirizzo di memoria che si trova oltre l'estensione dell'array, si potrebbe verificare un errore. Osservate inoltre che il secondo parametro delle due funzioni è di tipo **const char *** , cioè è una stringa costante. Entrambe le

funzioni, infatti, copiano questa stringa nel primo argomento senza mai modificarla. Applicando il principio del minimo privilegio, quindi, dichiariamo la stringa come costante. Nessuna delle due funzioni ha bisogno di modificare il secondo argomento per eseguire il proprio compito, per cui è preferibile non consentirne la modifica.

5.9 Gli array di puntatori

Gli elementi di un array possono essere a loro volta dei puntatori e sono utilizzati, ad esempio, per formare array di stringhe. Ogni elemento dell'array è una stringa, ma, come sapete già, una stringa è in sostanza un puntatore al suo primo carattere: perciò ogni elemento dell'array è nei fatti un puntatore al primo carattere di una stringa. Osservate la dichiarazione dell'array di stringhe `suit`, che contiene i nomi inglesi dei semi delle carte da gioco:

```
char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

La scrittura `suit[4]` indica un array di 4 elementi. La notazione `char *` indica invece che ogni elemento dell'array è un puntatore a un `char`. I quattro valori di inizializzazione dell'array sono `"Hearts"`, `"Diamonds"`, `"Clubs"` e `"Spades"` Ognuno di essi è memorizzato come una stringa terminata dal carattere nullo, per cui risulta più lunga di un carattere rispetto al numero di caratteri tra i doppi apici: le lunghezze delle stringhe sono, nell'ordine, di 7, 9, 6 e 7 caratteri. Anche se la notazione può far pensare che in `suit` sono memorizzate effettivamente le quattro stringhe, l'array contiene in realtà soltanto i puntatori al primo carattere di ogni stringa (Figura 5.22). In questo modo, anche se la dimensione dell'array `suit` è fissa, è possibile accedere a stringhe di qualsiasi lunghezza. Questa flessibilità è un esempio delle potenti capacità di strutturazione dei dati del C++. Le stringhe di `suit` potrebbero essere poste in un array bidimensionale, in cui ogni riga rappresenta un seme e ogni colonna una lettera del nome del seme. Una struttura dati di questo tipo dovrebbe avere un numero fisso di colonne per ogni riga, e questo numero dovrebbe prevedere la stringa più lunga possibile. D'altro canto utilizzando solo stringhe relativamente piccole si avrebbe un notevole spreco di spazio in memoria. Nella prossima sezione utilizzeremo un array di stringhe per rappresentare un mazzo di carte.

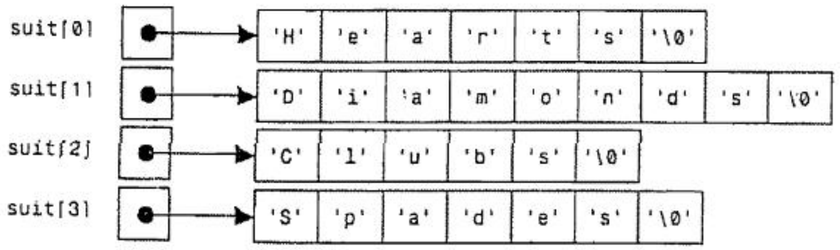


Figura 5.22 Rappresentazione grafica dell'array suit.

5.10 Un programma per mescolare e distribuire carte da gioco

In questa sezione utilizziamo i numeri casuali per scrivere un programma che mescola e distribuisce le carte da gioco. Esso può servire come punto di partenza di altri programmi che simulano giochi di carte. Vogliamo farvi familiarizzare con i problemi relativi all'efficienza, ed è per questo che abbiamo utilizzato intenzionalmente algoritmi *non* ottimali. Negli esercizi vi sarà chiesto di scrivere algoritmi più efficienti.

Utilizziamo il raffinamento top-down per scrivere il nostro programma, che deve mescolare un mazzo di 52 carte e distribuirle ai giocatori.

Rappresentiamo il mazzo di carte con l'array bidimensionale 4x13 `deck` (Figura 5.23). Le righe corrispondono ai semi: nella riga 0 abbiamo i cuori, nella riga 1 i quadri, nella riga 2 i fiori e nella riga 3 le picche. Le colonne corrispondono ai valori delle facce: dalla colonna 0 alla 9 abbiamo i valori dall'asso al 10, mentre dalla colonna 10 alla 12 abbiamo nell'ordine fante, donna e re. Dovremo porre nell'array di stringhe `suit` le stringhe che rappresentano i quattro semi e nell'array di stringhe `face` le stringhe che rappresentano i valori delle tredici facce.

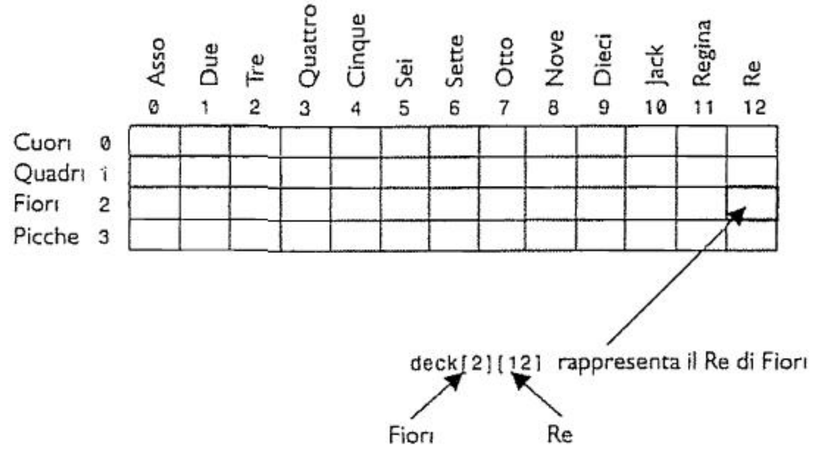


Figura 5.23 Rappresentazione del mazzo di carte tramite un array bidimensionale.

Come si fa a mescolare il mazzo di carte? Per prima cosa l'array `deck` (il mazzo) viene azzerato. Quindi sono scelti a caso `row` (0-3) e `column` (0-12); nell'elemento dell'array `deck[ row ][ column ]` viene inserito il numero 1, per indicare che questa sarà la prima carta a essere distribuita dopo il mescolamento. Questo procedimento continua con i numeri 2, 3 e così via, fino al numero 52. In questo modo avremo posto il mazzo in un particolare ordine, dalla prima alla cinquantaduesima carta. Man mano che l'array `deck` si riempie dei numeri posizionali, è possibile che una carta sia selezionata a caso una seconda volta, ovvero che `deck[ row ][ column ]` sia selezionata quando il suo valore è già diverso da zero. Se accade ciò, `deck[ row ][ column ]` viene ignorata e vengono generati casualmente altri due valori per `row` e `column`, fino a trovare una carta che non è stata già selezionata. Alla fine tutte le celle dell'array `deck` saranno occupate dai numeri 1-52 e a questo punto il mescolamento del mazzo è concluso.

Questo algoritmo di mescolamento può durare anche un tempo considerevole, specie se vengono scelte ripetutamente carte già selezionate. È un fenomeno noto con il nome di *differimento indefinito*. Negli esercizi elaboreremo un algoritmo migliore, per eliminare questa possibilità.

Obiettivo efficienza 5.3

A volte un algoritmo che sembra come la "naturale" soluzione al problema può nascondere dei problemi di efficienza difficili da individuare a prima vista: è il caso del differimento indefinito. Cercate di sviluppare algoritmi che non siano soggetti a questo genere di problemi.

Per la distribuzione della prima carta, dobbiamo cercare nell'array l'elemento `deck[ row ][ column ]` che vale 1. Lo facciamo con un costrutto `for` nidificato, in cui `row` varia da 0 a 3 e `column` da 0 a 12. Una volta trovato l'elemento, a che carta corrisponde? Nell'array `suit` sono stati immessi i quattro semi, per cui visualizziamo il seme che si trova nella stringa `suit[ row ]`. Allo stesso modo, per ottenere il valore della carta, visualizziamo la stringa `face[ column ]`.

Per completare la locuzione inglese scriviamo anche `"of"` (di): in questo modo scriveremo `"King of Clubs"`, `"Ace of Diamonds"` e così via.

Andiamo avanti adesso con il raffinamento top-down. Il top è semplice:

Mescolare e distribuire 52 carte

Il primo raffinamento porta a:

Inizializzare l'array suit
Inizializzare l'array face
Inizializzare l'array deck
Mescolare il mazzo
Distribuire 52 carte

Espandiamo la voce "Mescolare il mazzo" come segue:

Per ognuna delle 52 carte
    Assegnare un numero d'ordine in una cella a caso, non occupata, di deck

Espandiamo la voce "Distribuire 52 carte" come segue:

Per ognuna delle 52 carte
    Trovare il numero d'ordine della carta nell'array deck e visualizzare il suo valore e il suo seme

Incorporiamo queste piccole modifiche nel problema completo e avremo la nostra seconda ridefinizione:

Inizializzare l'array suit
Inizializzare l'array face
Inizializzare l'array deck

Per ognuna delle 52 carte
    Assegnare un numero d'ordine in una cella a caso, non occupata, di deck

Per ognuna delle 52 carte
    Trovare il numero d'ordine della carta nell'array deck e visualizzare il suo valore e il suo seme

Espandiamo la voce "Assegnare un numero d'ordine in una cella a caso, non occupata, di deck" come segue:

Scegliere una cella di deck in modo casuale

Se la cella scelta è già stata selezionata prima
    Scegliere una cella di deck in modo casuale

Assegnare alla cella di deck scelta un numero d'ordine

Espandiamo la voce "Trovare il numero d'ordine della carta nell'array deck e visualizzare il suo valore e il suo seme" come segue:

Per ogni cella dell'array deck
    Se la cella contiene un numero d'ordine
        Visualizzare il valore e il seme della carta

Incorporando queste modifiche abbiamo la nostra terza ridefinizione:

Inizializzare l'array suit
Inizializzare l'array face
Inizializzare l'array deck

Per ognuna delle 52 carte
    Scegliere una cella di deck in modo casuale

    Se la cella scelta è già stata selezionata prima
        Scegliere una cella di deck in modo casuale

    Assegnare alla cella di deck scelta un numero d'ordine

Per ognuna delle 52 carte
    Per ogni cella dell'array deck
        Se la cella contiene un numero d'ordine
        Visualizzare il valore e il seme della carta

Possiamo terminare qui il procedimento di ridefinizione. Notate che questo programma sarà più efficiente se la sezione che mescola le carte viene combinata con quella che le distribuisce, in modo che ogni carta venga data nell'ordine in cui si trova nel mazzo. Per il nostro esempio abbiamo scelto di separare le due sezioni perché normalmente le carte vengono distribuite ai giocatori soltanto dopo che il mazzo sia stato mescolato, e non contemporaneamente.

Il programma finale è in Figura 5.24 e il suo output è in Figura 5.25. Osservate la formattazione dell'output nella funzione `deal`:

```
cout << setw( 5 ) << setiosflags( ios::right )
     << wFace[ column ] << " of "
     << setw( 8 ) << setiosflags( ios::left )
     << wSuit[ row ]
     << ( card % 2 == 0 ? '\n'    '\t' );
```

Questa istruzione visualizza il valore della carta giustificato a destra in un campo di 5 caratteri, e il seme giustificato a sinistra in un campo di 8 caratteri. La visualizzazione si dispone su due colonne. Se la carta da visualizzare si trova nella prima colonna, viene inviato in output un carattere di tabulazione, in modo che la carta successiva si trovi nella seconda colonna. Altrimenti viene inviato in output un carattere di nuova linea.

```
1   // Fig. 5.24: fig05_24.cpp
2   // Programma che mescola e distribuisce carte da gioco
3   #include <iostream.h>
4   #include <iomanip.h>
5   #include <stdlib.h>
6   #include <time.h>
7
8   void shuffle( int [][ 13 ] );
9   void deal( const int [][ 13 ], const char *[], const char *[] );
10
11  int main()
12  {
13     const char *suit[ 4 ] =
14        { "Hearts", "Diamonds", "Clubs", "Spades" };
15     const char *face[ 13 ] =
16        { "Ace", "Deuce", "Three", "Four",
17          "Five", "Six", "Seven", "Eight",
18          "Nine", "Ten", "Jack", "Queen", "King" };
19     int deck[ 4 ][ 13 ] = { 0 };
20
21     srand( time( 0 ) );
22
23     shuffle( deck );
24     deal( deck, face, suit );
25
26     return 0;
27  }
28
29  void shuffle( int wDeck[][ 13 ] )
30  {
31     int row, column;
32
33     for ( int card = 1; card <= 52; card++ ) {
34        do {
35           row = rand() % 4;
36           column = rand() % 13;
37        } while( wDeck[ row ][ column ] != 0 );
38
39        wDeck[ row ][ column ] = card;
40     }
41  }
42
43  void deal( const int wDeck[][ 13 ], const char *wFace[],
44             const char *wSuit[] )
45  {
46     for ( int card = 1; card <= 52; card++ )
47
48        for ( int row = 0; row <= 3; row++ )
```

Figura 5.24 Programma che mescola e distribuisce le carte (continua)


```
49
50           for ( int column = 0; column <= 12; column++ )
51
52              if ( wDeck[ row ][ column ] == card )
53                 cout << setw( 5 ) << setiosflags( ios::right )
54                      << wFace[ column ] << " of "
55                      << setw( 8 ) << setiosflags( ios::left )
56                      << wSuit[ row ]
57                      << ( card % 2 == 0 ? '\n'   '\t' );
58  }
```

Figura 5.24 Programma che mescola e distribuisce le carte.

```
     Six of Clubs          Seven of Diamonds
     Ace of Spades           Ace of Diamonds
     Ace of Hearts         Queen of Diamonds
   Queen of Clubs          Seven of Hearts
     Ten of Hearts         Deuce of Clubs
     Ten of Spades         Three of Spades
     Ten of Diamonds        Four of Spades
    Four of Diamonds         Ten of Clubs
     Six of Diamonds         Six of Spades
   Eight of Hearts         Three of Diamonds
    Nine of Hearts         Three of Hearts
   Deuce of Spades           Six of Hearts
    Five of Clubs          Eight of Clubs
   Deuce of Diamonds       Eight of Spades
    Five of Spades          King of Clubs
    King of Diamonds        Jack of Spades
   Deuce of Hearts         Queen of Hearts
     Ace of Clubs           King of Spades
   Three of Clubs           King of Hearts
   Nine of Clubs           Nine of Spades
   Four of Hearts          Queen of Spades
   Eight of Diamonds        Nine of Diamonds
   Jack of Diamonds        Seven of Clubs
   Five of Hearts           Five of Diamonds
   Four of Clubs            Jack of Hearts
   Jack of Clubs           Seven of Spades
```

Figura 5.25 Output del programma che mescola e distribuisce le carte.

L'algoritmo per distribuire le carte ha però un punto debole: quando l'uguaglianza viene soddisfatta, anche se al primo tentativo, i due `for` interni continuano la ricerca sui restanti elementi di `deck`. Negli esercizi cercheremo di porre rimedio a questo difetto.

5.11 I puntatori a funzione

Un puntatore a una funzione contiene l'indirizzo di memoria di una funzione. Nel Capitolo 4 abbiamo visto che il nome di un array è in realtà l'indirizzo di memoria del primo suo elemento. Allo stesso modo il nome di una funzione è in realtà l'indirizzo di partenza del codice della funzione. I puntatori a funzioni possono essere passati alle funzioni, restituiti da funzioni, memorizzati in array e assegnati ad altri puntatori a funzioni.

Per mostrarvi come utilizzare questo tipo di puntatori abbiamo modificato il programma di ordinamento a bolle in Figura 5.15 in quello che vedete in Figura 5.26. La nuova versione è composta dalle funzioni `main`, `bubble`, `ascending` e `descending`. Un argomento della funzione `bubbleSort` è il puntatore a una funzione, che può essere o la funzione `ascending` o la funzione `descending`. Il programma chiede all'utente se vuole ordinare l'array in modo crescente o decrescente. Se l'utente immette 1, il programma passa alla funzione `bubble` il puntatore alla funzione `ascending` che ordina l'array in modo crescente. Se l'utente immette 2, il programma passa a `bubble` il puntatore a `descending`, che lo ordina in modo decrescente. L'output del programma è in Figura 5.27.

```
1   // Fig. 5.26: fig05_26.cpp   Programma di ordinamento
2   // generalizzato che utilizza i puntatori a funzione
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   void bubble( int [], const int, int (*)( int, int ) );
7   int ascending( int, int );
8   int descending( int, int );
9
10  int main()
11  {
12     const int arraySize = 10;
13     int order,
14         counter,
15         a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16
17     cout << "Enter 1 to sort in ascending order,\n"
18          << "Enter 2 to sort in descending order: ";
19     cin >> order;
20     cout << "\nData items in original order\n";
21
22     for ( counter = 0; counter < arraySize; counter++ )
23        cout << setw( 4 ) << a[ counter ];
24
25     if ( order == 1 ) {
26        bubble( a, arraySize, ascending );
27        cout << "\nData items in ascending order\n";
28     }
```

Figura 5.26 Programma per ordinare l'array in senso crescente o decrescente tramite i puntatori a funzioni (continua)


```
29     else {
30        bubble( a, arraySize, descending );
31        cout << "\nData items in descending order\n";
32     }
33
34     for ( counter = 0; counter < arraySize; counter++ )
35        cout << setw( 4 ) << a[ counter ];
36
37     cout << endl;
38     return 0;
39  }
40
41  void bubble( int work[], const int size,
42               int (*compare)( int, int ) )
43  {
44     void swap( int *, int * );
45
46     for ( int pass = 1; pass < size; pass++ )
47
48        for ( int count = 0; count < size - 1; count++ )
49
50           if ( (*compare)( work[ count ], work[ count + 1 ] ) )
51              swap( &work[ count ], &work[ count + 1 ] );
52  }
53
54  void swap( int *element1Ptr, int *element2Ptr )
55  {
56     int temp;
57
58     temp = *element1Ptr;
59     *element1Ptr = *element2Ptr;
60     *element2Ptr = temp;
61  }
62
63  int ascending( int a, int b )
64  {
65     return b < a;   // scambiare se b è minore di a
66  }
67
68  int descending( int a, int b )
69  {
70     return b > a;   // scambiare se b è maggiore di a
71  }
```

Figura 5.26 Programma per ordinare l'array in senso crescente o decrescente tramite i puntatori a funzioni.

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Figura 5.27 Output del programma di ordinamento a bolle in Figura 5.26.

L'intestazione di `bubble` contiene il parametro

```
int ( *compare )( int, int )
```

Questa notazione indica un puntatore a una funzione che riceve due parametri interi e restituisce un risultato intero. Le parentesi che racchiudono `*compare` sono obbligatorie, perché la precedenza di * è più bassa di quella delle parentesi che racchiudono i parametri della funzione. Senza le parentesi, la dichiarazione diventa

```
int *compare( int, int )
```

che dichiara una funzione che riceve due parametri interi e restituisce il puntatore a un intero.

Il nuovo prototipo della funzione `bubble` sarà

```
int ( * )( int, int )
```

Osservate che abbiamo scritto soltanto i tipi dei dati, ma volendo documentare meglio il programma è possibile includere anche i nomi corrispondenti, che il compilatore ignorerà.

La funzione passata a `bubble` viene chiamata in un'istruzione `if` come segue

```
if ( ( *compare )( work[ count ], work[ count + 1 ] ) )
```

Così come si deve dereferenziare un puntatore a una variabile per accedere al valore della variabile, si deve dereferenziare il puntatore a una funzione per poter eseguire la funzione.

Si può chiamare la funzione anche senza dereferenziare il puntatore, come in

```
if ( compare( work[ count ], work[ count + 1 ] ) )
```

che utilizza il puntatore direttamente come nome di funzione. Noi preferiamo il primo metodo di chiamata, perché evidenzia che `compare` è un puntatore a una funzione e quindi deve essere dereferenziato per accedere a essa. Il secondo metodo può generare confusione, perché fa pensare che `compare` è esso stesso una funzione.

Un utilizzo tipico dei puntatori a funzioni si trova nei sistemi di menu a scelta guidata. Si chiede all'utente di selezionare un'opzione tra le varie disponibili in un menu (per es. digitando un numero da 1 a 5) e a ciascuna opzione corrisponde una funzione che effettua quel particolare compito. I puntatori alle funzioni si trovano in un array di puntatori a funzioni; il numero digitato dall'utente diviene quindi l'indice di questo array e il puntatore dell'array scelto serve poi a chiamare la funzione deputata a quel compito.

In Figura 5.28 troviamo un esempio di programma che utilizza un array di puntatori a funzioni. Sono definite tre funzioni (`function1`, `function2` e `function3`); ognuna di esse prende un argomento intero e non restituisce alcun dato. L'array `f` contiene i puntatori a queste tre funzioni, ed è dichiarato in questo modo:

```
void ( *f[ 3 ] )( int ) = { function1, function2, function3 };
```

La dichiarazione si legge a partire dall'insieme di parentesi più a sinistra: "`f` è un array di 3 puntatori a funzioni che prendono ciascuna un argomento intero e restituiscono il tipo di dato `void`" L'array viene inizializzato con i nomi delle tre funzioni, che sono già tre puntatori. Quando l'utente immetterà un valore tra 0 e 2, quel valore verrà utilizzato come indice dell'array `f`. La funzione viene invocata così:

```
(*f[ choice ])( choice );
```

In questa chiamata, `f[ choice ]` seleziona il puntatore presente nell'array all'indirizzo `choice`. Il puntatore viene dereferenziato per chiamare la funzione, e `choice` viene passato come argomento della funzione chiamata. Ogni funzione visualizza il valore del suo argomento assieme al proprio nome, per indicare che è stata chiamata la funzione giusta. Negli esercizi svilupperemo un sistema di scelta multipla a menu.

```
1   // Fig. 5.28: fig05_28.cpp
2   // Esempio di utilizzo di un array di puntatori a funzione
3   #include <iostream.h>
4   void function1( int );
5   void function2( int );
6   void function3( int );
7
8   int main()
9   {
10     void (*f[ 3 ])( int ) = { function1, function2, function3 };
11     int choice;
12
13     cout << "Enter a number between 0 and 2, 3 to end: ";
14     cin >> choice;
15
16     while ( choice >= 0 && choice < 3 ) {
17        (*f[ choice ])( choice );
18        cout << "Enter a number between 0 and 2, 3 to end: ";
19        cin >> choice;
20     }
21
22     cout << "Program execution completed." << endl;
```

Figura 5.28 Uso di un array di puntatori a funzioni (continua)

```
23       return 0;
24    }
25
26    void function1( int a )
27    {
28       cout << "You entered " << a
29            << " so function1 was called\n\n";
30    }
31
32    void function2( int b )
33    {
34       cout << "You entered " << b
35            << " so function2 was called\n\n";
36    }
37
38    void function3( int c )
39    {
40       cout << "You entered " << c
41            << " so function3 was called\n\n";
42    }
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Figura 5.28 Uso di un array di puntatori a funzioni.

5.12 Introduzione alla manipolazione di caratteri e stringhe

In questa sezione introduciamo delle funzioni della libreria standard per la manipolazione delle stringhe. Le tecniche che discutiamo possono servire per lo sviluppo di editor di testo, word processor, software per l'impaginazione, sistemi tipografici computerizzati e altri tipi di software per l'elaborazione dei testi. Qui utilizzeremo le stringhe con la notazione puntatore. Nei capitoli successivi del libro tratteremo le stringhe come oggetti a tutti gli effetti.

5.12.1 Caratteri e stringhe: concetti fondamentali

I caratteri sono i mattoni fondamentali di un programma sorgente in C++: ogni programma si compone di una sequenza di caratteri che il computer interpreta come una serie di istruzioni per eseguire un compito specifico. Un programma può contenere delle costanti carattere: si tratta di un valore intero rappresentato come un carattere tra apici singoli. Il valore di una costante carattere è il numero intero corrispondente al carattere nel set di caratteri della macchina. Per esempio `'z'` rappresenta il valore intero di z, cioè 122 nel set di caratteri ASCII, mentre `'\n'` rappresenta il valore intero di nuova linea, cioè 10.

Una stringa è una serie di caratteri trattata come un'entità singola. Una stringa può includere lettere, numeri e vari caratteri speciali come +, -, *, / e $. Le cosiddette *costanti stringa* o *stringhe literal* sono sequenze di caratteri racchiusi tra doppi apici, come:

```
"John Q. Doe"                (un nome proprio)
"9999 Main Street"           (un indirizzo)
"Waltham, Massachusetts"     (una città e uno Stato)
"(201) 555-1212"             (un numero di telefono)
```

Una stringa è un array di caratteri che termina con il carattere nullo (`'\0'`). L'accesso ai caratteri di una stringa avviene tramite un puntatore al suo primo carattere. Il valore di una stringa è l'indirizzo costante del suo primo carattere: per questo motivo si può affermare correttamente che una stringa in C++ è un puntatore costante. In questo senso le stringhe sono, a tutti gli effetti, simili agli array, perché il nome di un array è anch'esso un puntatore costante al suo primo elemento.

Nelle dichiarazioni si può assegnare una stringa a un array di caratteri o a una variabile di tipo `char *`. Le dichiarazioni che seguono

```
char color[] = "blue";
char *colorPtr = "blue";
```

inizializzano le rispettive variabili con la stringa `"blue"`. La prima dichiarazione crea l'array di 5 elementi `color` che contiene i caratteri `'b'`, `'l'`, `'u'`, `'e'` e `'\0'`. La seconda crea la variabile puntatore `colorPtr` che punta all'indirizzo di memoria in cui il programma conserva la stringa `"blue"`.

Obiettivo portabilità 5.5

Quando si inizializza una variabile di tipo `char *` con una costante stringa, alcuni compilatori pongono la stringa in un indirizzo di memoria dove la stringa non può subire modifiche. Se il vostro programma, invece, ha bisogno di modificarla, dovreste conservarla in un array di caratteri, in modo da ottenere lo stesso comportamento su tutti i sistemi.

La dichiarazione `char color[] = { "blue" };` si può anche scrivere nella forma

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

Se dichiarate un array di caratteri e lo inizializzate con una stringa, l'array deve essere abbastanza esteso per conservare tutti i caratteri contenuti nella stringa, compreso il carattere nullo finale. La dichiarazione precedente determina automaticamente la dimensione dell'array sulla base del numero di inizializzatori forniti.

Errore tipico 5.15

Se non allocate spazio sufficiente in un array di caratteri, questo potrebbe non essere in grado di memorizzare un'intera stringa più il carattere nullo.

Errore tipico 5.16

Se create o utilizzate una stringa che non contiene il carattere terminatore state commettendo un errore.

Buona abitudine 5.5

Se memorizzate una stringa in un array di caratteri, assicuratevi che l'array sia abbastanza esteso da contenere la stringa più lunga che potreste memorizzare. In C++ si possono creare stringhe di qualsiasi lunghezza. Se una stringa viene copiata in un array di caratteri più piccolo della sua lunghezza, i caratteri finali della stringa andranno a sovrascrivere dati che si trovano nella memoria subito dopo le locazioni dell'array.

È possibile assegnare una stringa a un array utilizzando l'operatore di estrazione dallo stream e `cin`. Per esempio, l'istruzione che segue si può utilizzare per assegnare una stringa all'array di caratteri `word[20]`:

```
cin >> word;
```

La stringa immessa dall'utente viene memorizzata in `word`. Questa istruzione legge i caratteri dalla tastiera finché non incontra uno spazio, una tabulazione, un carattere di nuova linea o di fine del file (EOF). Notate che la stringa non dovrebbe superare i 19 caratteri, perché deve lasciare spazio per il carattere terminatore. È possibile utilizzare il manipolatore dello stream `setw`, di cui abbiamo parlato nel Capitolo 2, per assicurarsi che la stringa da leggere alla tastiera non superi le dimensioni dell'array `word`. Per esempio, l'istruzione

```
cin >> setw( 20 ) >> word;
```

specifica che `cin` deve leggere e memorizzare in `word` un massimo di 19 caratteri, per riservare in ogni caso la 20ma posizione dell'array al carattere terminatore della stringa. Il manipolatore dello stream `setw` viene applicato soltanto al successivo valore in lettura dall'input.

A volte si desidera leggere dall'input una linea intera di testo e memorizzarla in un array; per questo scopo il C++ fornisce la funzione `cin.getline`. Questa funzione prende i seguenti argomenti: un array di caratteri in cui verrà memorizzata la linea di testo, la lunghezza e un carattere detto delimitatore. Per esempio questo segmento di programma

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

dichiara l'array di 80 caratteri `sentence`, poi legge una linea di testo dalla tastiera e la memorizza nell'array. La funzione smette di leggere i caratteri in uno dei seguenti casi: quando incontra il delimitatore `'\n'`, quando incontra l'indicatore di fine file, o quando è stato raggiunto il numero massimo di caratteri specificati dal secondo parametro meno uno (l'ultimo carattere è riservato al carattere terminatore). Se la funzione incontra in input il carattere delimitatore, lo legge e lo scarta. Il terzo argomento di `cin.getline` ha come valore di default `'\n'`, per cui avremmo potuto scrivere la chiamata precedente nella forma

```
cin.getline( sentence, 80 );
```

Nel Capitolo 11 riprenderemo e approfondiremo la discussione su `cin.getline` e sulle altre funzioni per l'input e l'output.

Errore tipico 5.17

Se trattate un carattere singolo come una stringa potreste avere un errore fatale in fase di esecuzione. Una stringa, infatti, è un puntatore che, in genere, è associato ad un valore intero abbastanza grande. Un carattere singolo, invece, ha un valore intero molto piccolo: i valori dei caratteri del set ASCII vanno da 0 a 255. Su molti sistemi l'errore fatale si verifica perché le prime locazioni di memoria, a partire dall'indirizzo 0, sono riservate per scopi speciali, il più delle volte per i gestori di interrupt del sistema operativo. L'errore, quindi, si verifica per una violazione d'accesso.

Errore tipico 5.18

Se passate un carattere a una funzione, ma questa si aspetta una stringa, potreste avere un errore fatale in fase di esecuzione.

Errore tipico 5.19

Se passate una stringa a una funzione, ma questa si aspetta un carattere, commettete un errore di sintassi.

5.12.2 Le funzioni di libreria per le stringhe

La funzioni di libreria per le stringhe effettuano diverse operazioni, tra cui manipolazioni su stringhe, confronti tra due stringhe, ricerca di caratteri specifici in una stringa, segmentazione di una stringa in una serie di sezioni (token) logiche e determinazione della lunghezza di una stringa. In questa sezione presentiamo le più comuni funzioni di questo tipo; esse sono riepilogate in Figura 5.29.

Notate che molte delle funzioni in Figura 5.29 prevedono parametri di tipo `size_t`. Questo tipo di dato viene definito nel file di intestazione `<stddef.h>` che viene incluso automaticamente in altri file di intestazione, come `<string.h>` Il tipo `size_t` è definito come un tipo intero senza segno, equivalente a un `unsigned int` o a un `unsigned long` a seconda dei sistemi.

Errore tipico 5.20

Se dimenticate di includere il file di intestazione `<string.h>` e utilizzate funzioni di libreria per la manipolazione di stringhe, commettete un errore di sintassi.

Prototipo di funzione	Descrizione della funzione
`char *strcpy( char *s1, const char *s2 )`	Copia la stringa `s2` nell'array di caratteri `s1`. Restituisce il valore di `s1`.
`char *strcat( char *s1, const char *s2 )`	Concatena le stringhe `s1` e `s2` memorizzando il risultato in `s1`. Il primo carattere di `s2` sovrascrive il carattere terminatore di `s1`. Restituisce il valore della stringa `s1` modificata.

Figura 5.29 Funzioni di libreria per la manipolazione di stringhe (continua)

Prototipo di funzione	Descrizione della funzione
`char *strncpy( char *s1, const char *s2, size_t n )`	Copia al massimo n caratteri della stringa s2 nell'array di caratteri s1. Restituisce il valore di s1.
`char *strncat( char *s1, const char *s2, size_t n )`	Concatena al più n caratteri della stringa s2 con la stringa s1. Il primo carattere di s2 sovrascrive il carattere terminatore di s1. Restituisce il valore della stringa s1 modificata.
`int strcmp( const char *s1, const char *s2 )`	Confronta la stringa s1 con la stringa s2. Restituisce un valore uguale, minore o maggiore di 0, a seconda se s1 sia rispettivamente uguale, minore o maggiore di s2.
`int strncmp( const char *s1, const char *s2, size_t n )`	Confronta fino a n caratteri della stringa s1 con la stringa s2. Restituisce un valore uguale, minore o maggiore di 0, a seconda se s1 sia rispettivamente uguale, minore o maggiore di s2.
`char *strtok( char *s1, const char *s2 )`	Una sequenza di chiamate a `strtok` suddivide la stringa s1 in "token", ovvero in segmenti logici separati, come possono essere per esempio le parole in una linea di testo. I token sono separati dai caratteri contenuti nella stringa s2. La prima chiamata contiene come primo argomento s1, mentre tutte le successive devono essere effettuate con NULL come primo argomento. Per ogni chiamata la funzione restituisce un puntatore al token corrente. Se non ci sono ulteriori token nella stringa s1, la funzione restituisce NULL.
`size_t strlen( const char *s )`	Determina la lunghezza della stringa s: restituisce il numero di caratteri che precedono il carattere terminatore.

Figura 5.29 Funzioni di libreria per la manipolazione di stringhe.

La funzione `strcpy` copia il suo secondo argomento (una stringa) nel suo primo argomento (un array di caratteri). La funzione `strncpy` è equivalente a `strcpy`, eccetto per il fatto che copia solo il numero di caratteri specificati. Notate che la funzione `strncpy` non copia necessariamente il carattere terminatore del suo secondo argomento: questo viene copiato solo se il numero di caratteri specificato è maggiore della lunghezza della stringa. Per esempio, se il secondo argomento è `"test"`, la funzione copia anche il carattere terminatore solo se il suo terzo argomento vale almeno 5, cioè i 4 caratteri della parola `test` più il carattere terminatore. Se il terzo argomento è maggiore di 5, all'array vengono aggiunti tanti caratteri terminatori quanti ne occorrono per raggiungere tale numero.

Errore tipico 5.21

Se al primo argomento di `strncpy` non viene accodato un carattere terminatore, e il terzo argomento è minore o uguale alla lunghezza della stringa passata come secondo argomento, potreste avere un errore fatale in fase di esecuzione.

Il programma in Figura 5.30 fa uso di `strcpy` per copiare una stringa intera contenuta nell'array `x` nell'array `y`, e di `strncpy` per copiare i primi 14 caratteri dell'array `x` nell'array `y`. All'array `z` viene aggiunto un carattere terminatore (`'\0'`) perché la chiamata a `strncpy` non effettua automaticamente questa operazione (il terzo argomento è minore della lunghezza della stringa passata come secondo argomento).

```
1  // Fig. 5.30: fig05_30.cpp
2  // Utilizzo di strcpy e strncpy
3  #include <iostream.h>
4  #include <string.h>
5
6  int main()
7  {
8     char x[] = "Happy Birthday to You";
9     char y[ 25 ], z[ 15 ];
10
11    cout << "The string in array x is: " << x
12         << "\nThe string in array y is: " << strcpy( y, x )
13         << '\n';
14    strncpy( z, x, 14 );  // non copia il carattere nullo
15    z[ 14 ] = '\0';
16    cout << "The string in array z is: " << z << endl;
17
18    return 0;
19 }
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Figura 5.30 Le funzioni strcpy e strncpy.

La funzione `strcat` accoda il suo secondo argomento (una stringa) al primo argomento (un array di caratteri che contiene una stringa). Il primo carattere del secondo argomento rimpiazza il carattere nullo che termina la stringa passata come primo argomento. I programmatori devono accertarsi che l'array che utilizzano per contenere la prima stringa sia abbastanza esteso da contenere anche la combinazione delle due stringhe più il carattere terminatore. La funzione `strncat` accoda alla prima stringa il numero di caratteri specificato della seconda stringa, e accoda al risultato un carattere terminatore. Il programma in Figura 5.31 utilizza le funzioni `strcat` e `strncat`.

La Figura 5.32 confronta tre stringhe tramite le funzioni `strcmp` e `strncmp`. La prima confronta due stringhe carattere per carattere. Il suo risultato è uguale a 0 se le stringhe sono uguali, è un numero negativo se la prima stringa è minore della seconda ed è un numero positivo se la prima stringa è maggiore della seconda. La funzione `strncmp` è

equivalente a `strcmp`, eccetto il fatto che `strncmp` effettua il confronto solo sul numero di caratteri indicato. La funzione `strncmp` non confronta i caratteri che si trovano dopo un carattere *null*, in una stringa. Il programma visualizza i valori interi restituiti dalle chiamate di queste funzioni.

Errore tipico 5.22

Le funzioni `strcmp` e `strncmp` non restituiscono 1 quando i loro argomenti sono uguali: non lo dimenticate, o commetterete un errore logico. Le due funzioni restituiscono 0 (corrispondente al valore booleano `false`) quando trovano l'uguaglianza. Se in un test volete verificare se due stringhe sono uguali, quindi, dovrete confrontare il valore restituito da `strcmp` o `strncmp` con 0.

```
1   // Fig. 5.31: fig05_31.cpp
2   // Utilizzo di strcat e strncat
3   #include <iostream.h>
4   #include <string.h>
5
6   int main()
7   {
8      char s1[ 20 ] = "Happy ";
9      char s2[] = "New Year ";
10     char s3[ 40 ] = "";
11
12     cout << "s1 = " << s1 << "\ns2 = " << s2;
13     cout << "\nstrcat(s1, s2) = " << strcat( s1, s2 );
14     cout << "\nstrncat(s3, s1, 6) = " << strncat( s3, s1, 6 );
15     cout << "\nstrcat(s3, s1) = " << strcat( s3, s1 ) << endl;
16
17     return 0;
18  }
```

```
s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
```

Figura 5.31 Le funzioni strcat e strncat.

```
1   // Fig. 5.32: fig05_32.cpp
2   // Utilizzo di strcmp e strncmp
3   #include <iostream.h>
4   #include <iomanip.h>
5   #include <string.h>
6
7   int main()
8   {
9      char *s1 = "Happy New Year";
```

Figura 5.32 Le funzioni strcmp e strncmp (continua)


```
10     char *s2 = "Happy New Year";
11     char *s3 = "Happy Holidays";
12
13     cout << "s1 = " << s1 << "\ns2 = " << s2
14          << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
15          << setw( 2 ) << strcmp( s1, s2 )
16          << "\nstrcmp(s1, s3) = " << setw( 2 )
17          << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
18          << setw( 2 ) << strcmp( s3, s1 );
19
20     cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
21          << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
22          << setw( 2 ) << strncmp( s1, s3, 7 )
23          << "\nstrncmp(s3, s1, 7) = "
24          << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
25     return 0;
26  }
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Figura 5.32 Le funzioni strcmp e strncmp.

A questo punto è necessario chiarire cosa intendiamo quando diciamo che una stringa "è maggiore" o "è minore" di un'altra stringa. Questo concetto che è legato all'ordine *alfabetico di un elenco telefonico: se cercate un nome, sapete già che "Jones" si trova* prima di "Smith", perché la prima lettera di "Jones" nell'alfabeto viene prima della prima lettera di "Smith". Quindi la stringa "Jones" *è minore* della stringa "Smith". Nel caso di nomi con lettera iniziale uguale, come ad esempio "Jones" e "Jameson", per prendere una decisione si va a guardare la seconda lettera e così via. Questo tipo di ordinamento è detto anche *ordinamento lessicografico.*

Obiettivo portabilità 5.6

I codici numerici che rappresentano internamente i caratteri possono essere differenti su sistemi diversi.

Obiettivo portabilità 5.7

Non scrivete test che verificano direttamente l'uguaglianza di un carattere con un codice ASCII, come `if (ch == 65)`. Al suo posto potete utilizzare la costante carattere corrispondente, in questo caso `if (ch == 'A')`.

Le case produttrici di computer hanno compiuto uno sforzo per standardizzare le rappresentazioni dei caratteri, adottando uno tra i due schemi di codifica più diffusi, ASCII e EBCDIC. ASCII è l'acronimo di "American Standard Code for Information Interchange", mentre EBCDIC sta per "Extended Binary Coded Decimal Interchange Code"

ASCII e EBCDIC sono detti *codici di carattere* o *set di caratteri*. Le operazioni su stringhe e caratteri operano sui codici numerici dei caratteri, piuttosto che sui caratteri stessi. Tra l'altro questo vi aiuta a comprendere l'equivalenza tra caratteri e numeri interi in C++. *Così come ha un senso dire che un codice numerico è maggiore, uguale o minore di* un altro, è possibile dire la stessa cosa di due caratteri, o di due stringhe, sulla base dei valori numerici dei caratteri che contengono. Nell'appendice D trovate l'elenco dei codici ASCII dei caratteri.

La funzione `strtok` suddivide una stringa in una serie di *token*, cioè in sequenze di caratteri separate dal resto della stringa da caratteri detti *delimitatori*. Generalmente i delimitatori utilizzati sono gli spazi o i caratteri di punteggiatura. Per esempio, in una linea di testo ogni parola può essere considerata come un token, mentre gli spazi sono i delimitatori.

Per ottenere tutti i token contenuti in una stringa è necessario chiamare più volte la funzione `strtok` (sempre che la stringa ne contenga effettivamente più di uno). La prima chiamata a `strtok` deve passare la stringa da elaborare e una stringa contenente tutti i delimitatori che separano i token. Nel programma in Figura 5.33 l'istruzione

```
tokenPtr = strtok( string, " " );
```

assegna a `tokenPtr` un puntatore al primo token contenuto nella stringa. Il secondo argomento di `strtok`, `" "`, indica che i token della stringa sono separati da spazi. La funzione `strtok` ricerca il primo carattere della stringa che non è un delimitatore (uno spazio in questo caso) e questo diventa il primo carattere del primo token. Poi `strtok` ricerca nella stringa il delimitatore successivo e lo sostituisce con un carattere nullo (`'\0'`). Questa operazione termina il token corrente. La funzione conserva internamente un puntatore al carattere successivo della stringa e restituisce un puntatore al token trovato.

Per cercare i token successivi della stessa stringa, bisogna continuare a chiamare la funzione `strtok` con `NULL` come primo argomento. L'argomento `NULL` indica a `strtok` che deve continuare a cercare i token dal punto della stringa che è stato salvato durante la precedente esecuzione della funzione. Se `strtok` non trova alcun token, restituisce il valore `NULL`. Il programma in Figura 5.33 utilizza `strtok` per suddividere in token la frase inglese `"This is a sentence with 7 tokens"`. Ogni token viene visualizzato separatamente. Fate attenzione, perché `strtok` modifica la stringa di input: perciò, se prevedete di riutilizzare la stringa che passate a `strtok`, dovrete effettuarne una copia prima di chiamare la funzione.

Errore tipico 5.23

La funzione `strtok` modifica la stringa da suddividere in token: se lo dimenticate e riutilizzate la stringa dopo le chiamate a `strtok`, il programma assumerà un comportamento erroneo.


```
1  // Fig. 5.33: fig05_33.cpp
2  // Utilizzo di strtok
3  #include <iostream.h>
4  #include <string.h>
5
6  int main()
7  {
8     char string[] = "This is a sentence with 7 tokens";
9     char *tokenPtr;
10
11    cout << "The string to be tokenized is:\n" << string
12         << "\n\nThe tokens are:\n";
13
14    tokenPtr = strtok( string, " " );
15
16    while ( tokenPtr != NULL ) {
17       cout << tokenPtr << '\n';
18       tokenPtr = strtok( NULL, " " );
19    }
20
21    return 0;
22 }
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Figura 5.33 La funzione strtok.

La *funzione* `strlen` *riceve una stringa* come argomento e ne restituisce il numero di caratteri, escluso il carattere terminatore. Il programma in Figura 5.34 fa uso della funzione `strlen`.

```
1  // Fig. 5.34: fig05_34.cpp
2  // Utilizzo di strlen
3  #include <iostream.h>
4  #include <string.h>
5
6  int main()
7  {
8     char *string1 = "abcdefghijklmnopqrstuvwxyz";
```

Figura 5.34 La funzione strlen (continua)

```
 9      char *string2 = "four";
10      char *string3 = "Boston";
11
12      cout << "The length of \"" << string1
13           << "\" is " << strlen( string1 )
14           << "\nThe length of \"" << string2
15           << "\" is " << strlen( string2 )
16           << "\nThe length of \"" << string3
17           << "\" is " << strlen( string3 ) << endl;
18
19      return 0;
20   }
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Figura 5.34 La funzione strlen.

5.13 Pensare in termini di oggetti: le collaborazioni tra gli oggetti [progetto opzionale]

In questo capitolo termineremo la fase di progettazione del *simulatore di ascensore*, e dal prossimo inizieremo a programmarlo utilizzando il C++. Al termine di questa sezione vi forniamo un elenco di risorse in rete e una bibliografia su UML.

Discuteremo ora delle collaborazioni tra oggetti, dette anche interazioni. Quando due oggetti comunicano tra loro per portare a termine *un compito, si dice che essi collaborano*, questa attività è svolta inviando e ricevendo messaggi. Gli elementi fondamentali di una *collaborazione* sono: (i) l'oggetto che emette il messaggio, (ii) il tipo di messaggio inviato e (iii) l'oggetto che riceve il messaggio. Com'è usuale, il messaggio inviato dal primo oggetto invoca un'operazione sul secondo.

Nella sezione "Pensare in termini di oggetti" del Capitolo 4 abbiamo determinato diverse operazioni relative alle classi del nostro sistema; qui, invece, ci soffermeremo sui messaggi che invocano tali operazioni. La Figura 5.35 è la tabella delle classi e delle espressioni che abbiamo già incontrato nella Sezione 4.10, in cui abbiamo eliminato le espressioni che non corrispondono ad alcuna operazione. Abbiamo associato le espressioni "fornisce l'orario al programmatore" e "fornisce l'orario all'ascensore" alla classe `Building`, perché abbiamo deciso che sarà l'edificio a gestire il controllo della simulazione. Per lo stesso motivo abbiamo associato a `Building` anche le espressioni "incrementa l'orario" e "*ottiene l'orario*"

Per determinare le collaborazioni consideriamo i verbi: per esempio, alla classe `Elevator` è associata l'espressione "rilascia il pulsate interno" Per fare ciò, un oggetto della classe `Elevator` deve inviare il messaggio `resetButton` (rilascia pulsante) a un oggetto della classe `ElevatorButton`, invocando l'operazione `resetButton` di tale classe. La Figura 5.36 elenca tutte le collaborazioni che possiamo ricavare dalla tabella delle espressioni.

Classe	Espressioni
`Elevator`	rilascia il pulsante interno, suona il campanello, segnala il suo arrivo al piano, apre la porta, chiude la porta
`Clock`	emette un tic ad ogni istante
`Scheduler`	ordina a una persona di accedere al piano, verifica che il piano sia libero
`Person`	preme il pulsante esterno, preme il pulsante interno, entra nell'ascensore, esce dall'ascensore
`Floor`	rilascia il pulsante esterno, spegne la spia, accende la spia
`FloorButton`	chiama l'ascensore
`ElevatorButton`	segnala all'ascensore di prepararsi alla corsa
`Door`	(apertura) segnala alla persona di uscire dall'ascensore, (apertura) segnala alla persona di entrare nell'ascensore
`Bell`	
`Light`	
`Building`	incrementa l'orario, ottiene l'orario, fornisce l'orario al meccanismo di gestione degli arrivi, fornisce l'orario all'ascensore

Figura 5.35 Elenco (modificato) delle espressioni relative alle classi del sistema.

Un oggetto della classe	Invia un messaggio	A un oggetto della classe
`Elevator`	`resetButton`	`ElevatorButton`
	`RingBell`	`Bell`
	`elevatorArrived`	`Floor`
	`openDoor`	`Door`
	`closeDoor`	`Door`
`Clock`		
`Scheduler`	`stepOntoFloor`	`Person`
	`IsOccupied`	`Floor`
`Person`	`pressButton`	`FloorButton`
	`PressButton`	`ElevatorButton`
	`PassengerEnters`	`Elevator`
	`PassengerExits`	`Elevator`
	`PersonArrives`	`Floor`
	`Floor resetButton`	`FloorButton`
	`TurnOff`	`Light`
	`TurnOn`	`Light`
`FloorButton`	`summonElevator`	`Elevator`
`ElevatorButton`	*`prepareToLeave`*	*`Elevator`*
`Door`	`exitElevator`	`Person`
	`EnterElevator`	`Person`

Figura 5.36 Le collaborazioni nel sistema ascensore (continua)

Un oggetto della classe	Invia un messaggio	A un oggetto della classe
`Bell`		
`Light`		
`Building`	`tick`	`Clock`
	`GetTime`	`Clock`
	`ProcessTime`	`Scheduler`
	`ProcessTime`	`Elevator`

Figura 5.36 Le collaborazioni nel sistema ascensore.

5.13.1 I diagrammi delle collaborazioni

Esaminiamo ora le interazioni tra oggetti che consentiranno alle persone della simulazione di entrare e uscire dall'ascensore, quando questo arriva al piano. UML fornisce uno strumento chiamato *diagramma delle collaborazioni* che permette di rappresentare queste interazioni. I diagrammi delle collaborazioni e di sequenza rappresentano entrambi il modo in cui interagiscono gli oggetti, ma ciascuno di essi si sofferma su un aspetto particolare. Nei diagrammi di sequenza si pone l'accento sul *momento* in cui avvengono le interazioni, mentre nei diagrammi delle collaborazioni su quali oggetti vi *partecipano*.

La Figura 5.37 rappresenta un diagramma delle collaborazioni che raccoglie le interazioni tra gli oggetti quando le persone entrano ed escono dall'ascensore. La collaborazione inizia quando l'ascensore arriva a un piano. Come nel caso dei diagrammi di sequenza, anche nei diagrammi delle collaborazioni un oggetto è rappresentato da un rettangolo che ne contiene il nome.

Gli oggetti coinvolti nella collaborazione sono collegati da linee continue su cui transitano i messaggi che essi si scambiano, nella direzione indicata dalle frecce. Il nome del messaggio è indicato vicino alla freccia.

La *sequenza dei messaggi*, in un diagramma delle collaborazioni, procede secondo una numerazione crescente. L'ascensore invia il primo messaggio (indicato dal numero 1 e di nome `resetButton`) al pulsante interno, per rilasciarlo, quindi invia il messaggio `ringBell` (messaggio n. 2) al campanello ed infine notifica il suo arrivo al piano (messaggio n. 3), in modo che il piano possa rilasciare il pulsante esterno e accendere la spia (messaggi 3.1 e 3.2).

Non appena il piano ha compiuto queste operazioni, l'ascensore apre la porta (messaggio 4) e, a questo punto, la porta invia il messaggio `exitElevator` (messaggio 4.1) all'oggetto `passenger` (Nota: Nel mondo reale, la persona nell'ascensore deve aspettare l'apertura della porta prima di poter scendere e vogliamo rappresentare fedelmente questo comportamento. Perciò, nel nostro modello la porta invia un messaggio all'oggetto `passenger` nell'ascensore. Esso può essere immaginato come un segnale visivo per la persona nell'ascensore che gli indica l'apertura della porta.). A sua volta, l'oggetto `passenger` comunica all'ascensore il suo intento di uscire tramite il messaggio `passengerExits` (messaggio 4.1.1).

Dopo l'uscita del passeggero, la persona in attesa al piano (l'oggetto `waitingPassenger`) può entrare nell'ascensore. Notate che la porta invia il messaggio `enterElevator` (messaggio 4.2) all'oggetto `waitingPassenger`, dopo che l'oggetto `passenger` ha comunicato

all'ascensore la volontà di uscire attraverso il messaggio `passengerExits` (messaggio 4.1.1). Questa sequenza assicura che la persona al piano aspetti l'uscita del passeggero dall'ascensore, prima di entrarvi. L'oggetto `waitingPassenger` entra nell'ascensore tramite il messaggio `passengerEnters` (messaggio 4.2.1).

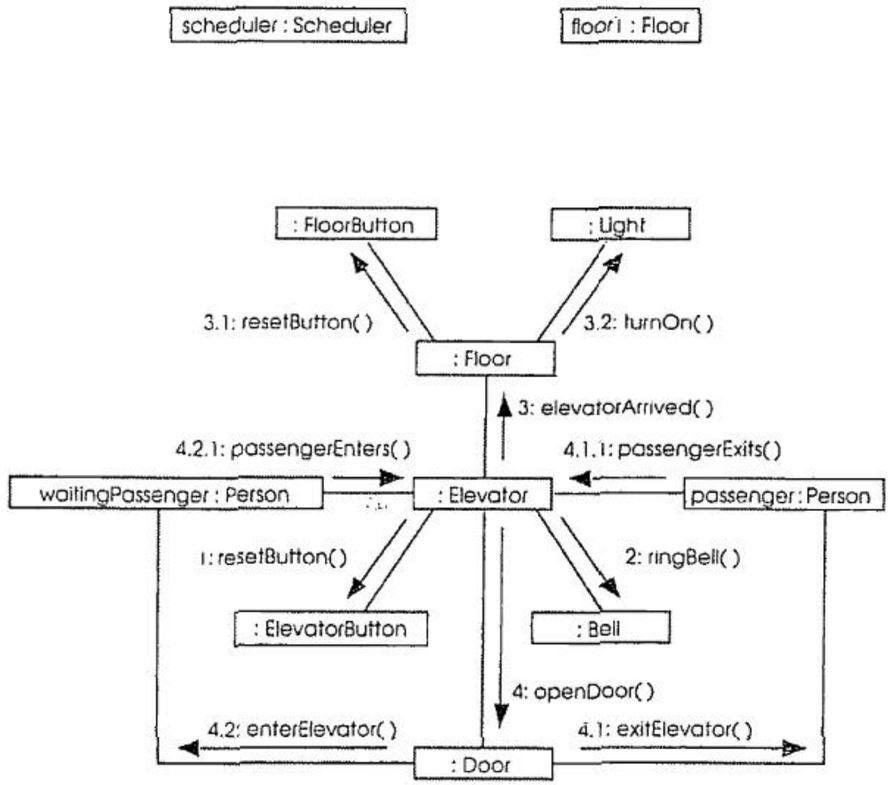


Figura 5.37 Diagramma delle collaborazioni per caricare e scaricare i passeggeri.

5.13.2 Riepilogo

A questo punto abbiamo una comprensione sufficiente delle classi da implementare e delle interazioni tra gli oggetti. Dal prossimo capitolo inizieremo a studiare la programmazione orientata agli oggetti in C++, per cui potremo scrivere una porzione significativa di codice del simulatore. Dal Capitolo 7 potremo implementare un simulatore completo e funzionante, mentre nel Capitolo 9 introdurremo l'ereditarietà, che consente di sfruttare le caratteristiche comuni delle classi per ridurre al minimo il codice da implementare.

Riepiloghiamo ora le fasi di progettazione che abbiamo iniziato dal Capitolo 2.

0. Nella fase di analisi parlate con i clienti che vi hanno commissionato la scrittura del software, raccogliendo più informazioni possibili sulle sue caratteristiche. Tali informazioni servono per definire i casi d'uso, che descrivono il modo in cui gli utenti interagiranno con il sistema. Nell'esposizione abbiamo saltato questa fase riportandone direttamente il risultato, cioè la definizione del problema e i casi d'uso. Vi ricordiamo che i sistemi reali presentano anche parecchi casi d'uso.

1. Individuate le classi del sistema elencando i nomi presenti nella definizione del problema. Eliminate dall'elenco i nomi che rappresentano in modo chiaro gli attributi delle classi e quelli *che evidentemente non sono parte del sistema*. Create un diagramma delle classi che rappresenti le classi del sistema e le loro relazioni reciproche (associazioni).
2. Individuate gli attributi di ogni classe dalla definizione del problema elencando le parole e le frasi che la descrivono.
3. Cercate di comprendere al meglio la dinamica del sistema. Create i diagrammi degli stati per capire come le classi del sistema mutano nel tempo.
4. Prendete in esame verbi ed espressioni associati a ciascuna classe e individuate tramite essi le operazioni. I diagrammi delle attività sono *utili per rappresentare i dettagli delle* operazioni.
5. Esaminate le collaborazioni tra i vari oggetti e rappresentatele nei diagrammi di sequenza e delle collaborazioni. Aggiungete attributi e operazioni alle classi, man mano che la progettazione le richiede.
6. A questo punto la progettazione contiene ancora delle lacune che risulteranno evidenti non appena implementerete il simulatore in C++, a partire dal prossimo Capitolo.

5.13.3 Risorse in rete sull'UML

Segue un elenco di risorse in rete sull'UML, tra cui documenti sulle specifiche UML versione 1.3 e altri materiali di riferimento, *risorse generali, tutorial, FAQ, articoli, temi* e software.

Riferimenti

`www.omg.org`

È il sito ufficiale dell'OMG (Object Management Group), un organizzazione che studia *le metodologie alla base della programmazione* ad oggetti, in generale, e di UML, in particolare, e che ne supervisiona la manutenzione e le future revisioni. Il sito web contiene informazioni su UML e su altre tecnologie orientate agli oggetti.

`www.rational.com`

UML è stato sviluppato inizialmente da "Rational Software Corporation". Il sito web contiene informazioni su UML e sulle persone che lo hanno progettato: Grady Booch, James Rumbaugh e Ivar Jacobson.

`www.omg.org/cgi-bin/doc?ad/99-06-09`

Contiene versioni in formato PDF e ZIP delle specifiche UML versione 1.3.

`www.omg.org/techprocess/meetings/schedule/UML_1.4_RTF.html`

L'OMG mantiene su questa pagina informazioni sulle specifiche della futura versione 1.4 di UML.

`www.rational.com/uml/resources/quick/index.jtmpl`

Breve guida su UML.

`www.holub.com/class/oo_design/uml.html`

Questo sito fornisce un breve manuale di riferimento su UML e delle note illustrative.

`www.softdocwiz.com/UML.htm`

In questo sito Kendall Scott mantiene un dizionario su UML.

Risorse

`www.omg.org/uml/`

Risorsa su UML di OMG.

`www.rational.com/uml/index.jtmpl`

Risorsa su UML di "Rational Software Corporation"

`www.platinum.com/corp/uml/uml.htm`

"Platinum Technology", membro dei Partner UML, mantiene una pagina di risorse su UML a questo indirizzo.

`www.cetus-links.org/oo_uml.html`

Contiene centinaia di link a siti aventi UML come argomento, dove si possono trova*re informazioni, tutorial e software.*

`www.uml-zone.com`

Questo sito è ricco di informazioni su UML e include articoli e link a newsgroup e altri siti.

`home.pacbell.net/ckobryn/uml.htm`

Sito mantenuto da Cris Kobryn: Contiene informazioni generali e link a siti web importanti.

`www.methods-tools.com/cgi-bin/DiscussionUML.cgi`

Contiene un gruppo di discussione su UML.

`www.pols.co.uk/usecasezone/index.htm`

Il sito fornisce risorse ed articoli sui casi operativi.

`www.ics.uci.edu/pub/arch/uml/uml_books_and_tools.html`

Il sito contiene link per raccogliere informazioni su testi su UML e un elenco di strumenti di sviluppo che supportano la notazione UML.

`home.earthlink.net/~salhir/`

Sinan Si Alhir, autore del testo "UML in a Nutshell", mantiene un sito che include link a diverse risorse su UML.

Software

`www.rational.com/products/rose/index.jtmpl`

Home page di "Rational Rose™", lo strumento di sviluppo visuale di "Rational Software Corporation". È possibile scaricarne una versione demo ed utilizzarla per un periodo limitato.

`www.rosearchitect.com/`

"Rosearchitect.com" è una rivista elettronica pubblicata da "Rational Software Corporation" che tratta di UML e di "Rational Rose".

`www.advancedsw.com/`

"Advanced Software Technologies" produce GDPro: uno strumento di sviluppo visuale UML. È possibile scaricarne una versione demo ed utilizzarla per un periodo limitato.

`www.visualobject.com/`

"Visual Object Modelers" ha creato uno strumento di sviluppo visuale UML. È possibile scaricarne una versione demo ed utilizzarla per un periodo limitato.

`www.microgold.com/version2/stage/product.html`

"Microgold Software, Inc" ha creato "WithClass", un'applicazione di progettazione software che supporta la notazione UML.

`www.lysator.liu.se/~alla/dia/dia.html`

"Dia" è uno programma per la creazione di diagrammi basato sulla libreria gtk+, fra le altre caratteristiche, esso supporta gli strumenti per i diagrammi delle classi dell'UML. Dia è progettato per sistemi UNIX, ma il sito contiene un link a una versione Windows del programma.

`dir.lycos.com/Computers/Software/Object_Oriented/Methodologies/UML/Tools/`

Il sito elenca decine di strumenti di sviluppo UML, con le rispettive home page.

`www.methods-tools.com/tools/modeling.html`

Il sito contiene un elenco di diversi strumenti di sviluppo orientati agli oggetti, tra cui alcuni che supportano UML.

Articoli e temi

`www.omg.org/news/pr99/UML_2001_CACM_Oct99_p29-Kobryn.pdf`

Scritto da Cris Kobryn, esplora il passato, il presente e il futuro di UML.

`www.sdmagazine.com/uml/focus.rosenberg.htm`

Parla di come utilizzare UML nei propri progetti.

`www.db.informatik.uni-bremen.de/umlbib/`

Bibliografia su UML, include nomi e autori di diversi articoli sull'argomento. È possibile effettuare una ricerca per autore o per titolo.

`usecasehelp.com/wp/white_papers.htm`

Elenco di articoli sull'applicazione dei casi operativi all'analisi e alla progettazione di un sistema.

`www.ratio.co.uk/white.html`

Articolo che delinea il procedimento di analisi e progettazione ad oggetti (OOAD) utilizzando UML. C'è anche una parte sull'implementazione in C++.

`www.tucs.fi/publications/techreports/TR234.pdf`

Studio tramite UML di un caso OOAD relativo a un registratore digitale.

`www.conallen.com/whitepapers/webapps/ModelingWebApplications.htm`

Contiene lo studio di un caso di sviluppo di applicazioni Web tramite UML.

`www.sdmagazine.com/`

"Software Development Magazine Online" è una miniera di articoli su UML. È possibile effettuare una ricerca per autore o per titolo.

Tutorial

`www.qoses.com/education/`

Database di tutorial creato da Kendall Scott e mantenuto da Qoses.

`www.qoses.com/education/tests/test02.html`

Quiz su UML i cui risultati vengono spediti per posta elettronica.

`www.rational.com/products/rose/tryit/tutorial/index.jtmpl`

Tutorial di "Rational Rose"

FAQ

`www.rational.com/uml/gstart/faq.jtmpl`

FAQ su UML di "Rational Software Corporation"

`usecasehelp.com/faq/faq.htm`

Breve FAQ mantenuto da "usecasehelp.com".

`www.jguru.com/jguru/faq/`

Per accedere al FAQ su UML bisogna indicare UML nella casella di testo.

`www.uml-zone.com/umlfaq.asp`

Breve FAQ mantenuto da "uml-zone.com"

Esercizi di autovalutazione

5.1 Completate le seguenti affermazioni:

a) Un puntatore è una variabile che contiene il valore dell' _____ di un'altra variabile.
b) I tre valori che possono essere utilizzati per inizializzare un puntatore sono _____, _____ o _____.
c) L'unico valore intero che può essere assegnato a un puntatore è _____.

5.2 Determinate quali sono le affermazioni vere. Per quelle false, spiegatene il motivo.

a) L'operatore indirizzo `&` si può applicare soltanto a costanti, espressioni o variabili della classe di memorizzazione register.
b) Un puntatore dichiarato come void può essere dereferenziato.
c) I puntatori di tipo diverso non possono essere assegnati l'uno all'altro senza un'operazione di cast.

5.3 Rispondete alle domande. Per tutto l'esercizio supporremo che un numero a virgola mobile e singola precisione occupa 4 byte di memoria, e che l'indirizzo di partenza dell'array sia alla posizione di memoria 1.002.500. Dove serve, nell'esercizio utilizzate i risultati ricavati dalle risposte precedenti.

a) Dichiarate un array di tipo `float` di nome `numbers`, contenente 10 elementi, e inizializzate gli elementi con i valori 0.0, 1.1, 2.2, ..., 9.9. Supponete che la costante simbolica `SIZE` sia stata definita uguale a `10`.
b) Dichiarate il puntatore `nPtr`, che punta a un oggetto di tipo `float`.
c) Visualizzate gli elementi dell'array `numbers` usando la notazione con gli indici. Utilizzate un costrutto `for`, supponendo di aver già definito altrove la variabile di controllo `i`. Visualizzate ogni numero con la precisione di 1 numero a destra del punto decimale.
d) Scrivete due istruzioni distinte che assegnano l'indirizzo di partenza dell'array `numbers` al puntatore `nPtr`.
e) Visualizzate gli elementi di `numbers` utilizzando la notazione puntatore/offset con il puntatore `nPtr`.
f) Visualizzate gli elementi di `numbers` utilizzando la notazione puntatore/offset con il nome dell'array come puntatore.
g) Visualizzate gli elementi di `numbers` indicizzando il puntatore `nPtr`.
h) Riferite l'elemento 4 dell'array `numbers` utilizzando la notazione con gli indici, la notazione puntatore/offset con il nome dell'array come puntatore, indicizzando il puntatore `nPtr` e con la notazione puntatore/offset con il puntatore `nPtr`.
i) Supponendo che `nPtr` punti alla posizione iniziale dell'array `numbers`, che indirizzo riferisce `nPtr + 8`? Che valore occupa quella locazione?
j) Supponendo che `nPtr` punti a `numbers[5]`, che indirizzo riferisce `nPtr` dopo l'esecuzione dell'istruzione `nPtr -= 4`? Che valore occupa quella locazione?

5.4 Per ogni punto di questo esercizio scrivete un'istruzione che svolga il compito indicato. Supponete di aver dichiarato le variabili a virgola mobile `number1` e `number2`, e che `number1` sia stato inizializzato a `7.3`. Supponete anche che la variabile `ptr` sia di tipo `char *` e che gli array `s1[ 100 ]` e `s2[ 100 ]` siano di tipo `char`.

a) Dichiarate la variabile `fPtr` come puntatore a un oggetto di tipo `float`.
b) Assegnate l'indirizzo della variabile `number1` al puntatore `fPtr`.
c) Visualizzate il valore dell'oggetto a cui punta `fPtr`.
d) Assegnate il valore dell'oggetto a cui punta `fPtr` alla variabile `number2`.
e) Visualizzate il valore di `number2`.
f) Visualizzate l'indirizzo di `number1`.
g) Visualizzate l'indirizzo contenuto in `fPtr`. Il valore visualizzato è uguale a quello dell'indirizzo di `number1`?

h) Copiate nell'array `s1` la stringa contenuta nell'array `s2`.
i) Confrontate la stringa contenuta in `s1` con la stringa contenuta in `s2`. Visualizzate il risultato.
j) Accodate 10 caratteri della stringa in `s2` alla stringa in `s1`.
k) Determinate la lunghezza della stringa contenuta in `s1`. Visualizzate il risultato.
l) Assegnate a `ptr` la posizione del primo token contenuto in `s2`. I tokens di `s2` sono separati da virgole (`,`).

5.5 Eseguite le seguenti operazioni:

a) Scrivete l'intestazione della funzione `exchange`, che riceve come parametri `x` e `y`, due puntatori a numeri in virgola mobile, e non restituisce alcun valore.
b) Scrivete il prototipo di funzione di `exchange`.
c) Scrivete l'intestazione della funzione `evaluate`, che restituisce un numero intero e prende come parametri l'intero `x` e il puntatore alla funzione `poly`. La funzione `poly` riceve un parametro intero e restituisce un valore intero.
d) Scrivete il prototipo di funzione di `evaluate`.
e) Inizializzate l'array di caratteri `vowel` con la stringa `"AEIOU"` in due modi diversi.

5.6 Trovate l'errore nel segmento di programma che vi proponiamo. Supponiamo che

```
int *zPtr;  // zPtr riferirà l'array z
int *aPtr = 0;
void *sPtr = 0;
int number, i;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
sPtr = z;
```

a) `++zptr;`
b) `// usa il puntatore per ottenere il primo valore dell'array`
   `number = zPtr;`
c) `// assegna l'elemento di indice 2 dell'array (il valore 3) a number`
   `number = *zPtr[ 2 ];`
d) `// stampa l'intero array z`
   `for ( i = 0; i <= 5; i++ )`
      `cout << zPtr[ i ] << endl;`
e) `// assegna il valore puntato da sPtr a number`
   `number = *sPtr;`
f) `++z;`
g) `char s[ 10 ];`
   `cout << strncpy( s, "hello", 5 ) << endl;`
h) `char s[ 12 ];`
   `strcpy( s, "Welcome Home" );`
i) `if ( strcmp( string1, string2 ) )`
      `cout << "The strings are equal" << endl;`

5.7 Nell'esecuzione delle seguenti istruzioni viene visualizzato qualcosa? Se sì, cosa? Se un'istruzione contiene un errore, descrivete l'errore e indicate come rimediare. Supponiamo di aver dichiarato queste variabili:

```
char s1[ 50 ] = "jack", s2[ 50 ] = "jill", s3[ 50 ], *sptr;
```

a) `cout << strcpy( s3, s2 ) << endl;`
b) `cout << strcat( strcat( strcpy( s3, s1 ), " and " ), s2 )`
   `<< endl;`
c) `cout << strlen( s1 ) + strlen( s2 ) << endl;`
d) `cout << strlen( s3 ) << endl;`

Risposte agli esercizi di autovalutazione

5.1 a) indirizzo. b) `0`, `NULL`, un indirizzo. c) `0`.

5.2 a) Falso. L'operatore indirizzo si può applicare soltanto alle variabili, e non alle costanti, alle espressioni o alle variabili della classe di memorizzazione `register`.
b) Falso. Un puntatore al tipo `void` non può essere dereferenziato, perché il compilatore non ha modo di determinare quanti byte di memoria dereferenziare.
c) Falso. A puntatori di tipo `void` possono essere assegnati puntatori di tipo diverso. I puntatori di tipo `void`, però, non possono essere assegnati a puntatori di altro tipo senza un cast esplicito.

5.3 a)
```
float numbers[ SIZE ] = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
```
b)
```
float *nPtr;
```
c)
```
cout << setiosflags( ios::fixed | ios::showpoint )
     << setprecision( 1 );
for ( i = 0; i < SIZE; i++ )
   cout << numbers[ i ] << ' ';
```
d)
```
nPtr = numbers;
nPtr = &numbers[ 0 ];
```
e)
```
cout << setiosflags( ios::fixed | ios::showpoint )
     << setprecision( 1 );
for ( i = 0; i < SIZE; i++ )
   cout << *( nPtr + i ) << ' ';
```
f)
```
cout << setiosflags( ios::fixed | ios::showpoint )
     << setprecision( 1 );
for ( i = 0; i < SIZE; i++ )
   cout << *( numbers + i ) << ' ';
```
g)
```
cout << setiosflags( ios::fixed | ios::showpoint )
     << setprecision( 1 );
for ( i = 0; i < SIZE; i++ )
   cout << nPtr[ i ] << ' ';
```
h)
```
numbers[ 4 ]
*( numbers + 4 )
nPtr[ 4 ]
*( nPtr + 4 )
```
i) L'indirizzo è `1002500 + 8 * 4 = 1002532`. Il valore è `8.8`.
j) L'indirizzo di `numbers[ 5 ]` è `1002500 + 5 * 4 = 1002520`.
L'indirizzo di `nPtr -= 4` è `1002520 - 4 * 4 = 1002504`.
Il valore contenuto in quella posizione è `1.1`.

5.4 a)
```
float *fPtr;
```
b)
```
fPtr = &number1;
```
c)
```
cout << "The value of *fPtr is " << *fPtr << endl;
```
d)
```
number2 = *fPtr;
```
e)
```
cout << "The value of number2 is " << number2 << endl;
```
f)
```
cout << "The address of number1 is " << &number1 << endl;
```
g)
```
cout << "The address stored in fPtr is " << fPtr << endl;
```
Sì, il valore è uguale.
h)
```
strcpy(s1, s2);
```
i)
```
cout << "strcmp(s1, s2) = " << strcmp( s1, s2 ) << endl;
```
j)
```
strncat( s1, s2, 10 );
```
k)
```
cout << "strlen(s1) = " << strlen( s1 ) << endl;
```
l)
```
ptr = strtok( s2, "," );
```


5.5 a)
```
void exchange( float *x, float *y )
```
b)
```
void exchange( float *, float * );
```
c)
```
int evaluate( int x, int (*poly)( int ) )
```
d)
```
int evaluate( int, int (*)( int ) );
```
e)
```
char vowel[] = "AEIOU";
char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };
```

5.6 a) Errore: `zPtr` non è stato inizializzato.
Correzione: Inizializzate `zPtr` con `zPtr = z;`
b) Errore: Il puntatore non è stato dereferenziato.
Correzione: Modificate l'istruzione in `number = *zPtr;`
c) Errore: `zPtr[ 2 ]` non è un puntatore e non deve essere dereferenziato.
Correzione: Modificate `*zPtr[ 2 ]` in `zPtr[ 2 ]`.
d) Errore: Si sta riferendo un elemento oltre i limiti dell'array.
Correzione: Modificate l'operatore relazionale nel costrutto `for` in `<` per evitare di oltrepassare i limiti dell'array.
e) Errore: Si sta dereferenziando un puntatore `void`.
Correzione: Per dereferenziare questo puntatore dovete prima effettuarne la conversione ad un tipo intero. Modificate l'istruzione in `number = *(int *)sPtr;`
f) Errore: Si sta cercando di modificare il nome di un array con l'aritmetica dei puntatori.
Correzione: Utilizzate una variabile puntatore al posto del nome dell'array, per mantenere le operazioni di aritmetica dei puntatori così come sono, o indicizzate il nome dell'array per riferire un elemento specifico.
g) Errore: La funzione `strncpy` non scrive un carattere terminatore nell'array `s` perché il suo terzo argomento coincide con la lunghezza della stringa `"hello"`
Correzione: Come terzo argomento di `strncpy` passate `6` o assegnate `'\0'` a `s[ 5 ]` per assicurarvi che il carattere terminatore sia effettivamente scritto alla fine della stringa.
h) Errore: L'array di caratteri `s` non è abbastanza esteso per contenere il carattere terminatore.
Correzione: Aumentate gli elementi dell'array.
i) Errore: La funzione `strcmp` restituisce il valore 0 se le stringhe sono uguali; di conseguenza la condizione del costrutto `if` sarà falsa e l'istruzione di output verrà saltata.
Correzione: Confrontate esplicitamente il risultato di `strcmp` con 0 nella condizione del costrutto `if`.

5.7 a) `jill`
b) `jack and jill`
c) `8`
d) `13`

Esercizi

5.8 Indicate le affermazioni vere e quelle false. Per quelle false, indicatene il motivo.
a) Non ha senso confrontare due puntatori che puntano ad array diversi.
b) Dato che il nome di un array è un puntatore al suo primo elemento, i nomi degli array possono essere manipolati esattamente come i puntatori ordinari.

5.9 Rispondete alle seguenti domande. Supponete che gli interi di tipo `unsigned` occupino 2 byte e che l'indirizzo di partenza dell'array sia alla posizione di memoria 1002500.
a) Dichiarate l'array `values`, di tipo `unsigned int` e di 5 elementi, e inizializzate gli elementi con i numeri pari da 2 a 10. Supponete che la costante simbolica `SIZE` sia stata definita uguale a 5.

b) Dichiarate `vPtr`, puntatore a oggetti di tipo `unsigned int`.
c) Visualizzate gli elementi dell'array `values` indicizzando il nome dell'array. Utilizzate un costrutto `for`, supponendo di aver già dichiarato altrove la variabile di controllo `i`.
d) Scrivete due istruzioni distinte che assegnano l'indirizzo di partenza dei valori dell'array al puntatore `vPtr`.
e) Visualizzate gli elementi dell'array utilizzando la notazione puntatore /offset.
f) Visualizzate gli elementi dell'array utilizzando la notazione puntatore/offset con il nome dell'array come puntatore.
g) Visualizzate gli elementi dell'array indicizzando il puntatore all'array.
h) Referenziate il valore dell'elemento 5 utilizzando la notazione con gli indici e il nome dell'array, la notazione puntatore/offset con il nome dell'array come puntatore, la notazione con gli indici e il puntatore e la notazione puntatore/offset.
i) Che indirizzo riferisce `vPtr + 3`? Che valore occupa quella locazione?
j) Supponendo che `vPtr` punti a `values[ 4 ]`, che indirizzo riferisce `vPtr -= 4`? Che valore occupa quella locazione?

5.10 Scrivete un'istruzione per ognuno dei compiti elencati. Supponete di aver dichiarato `value1` e `value2` come variabili `long int` e di aver inizializzato `value1` con il valore `200000`.

a) Dichiarate la variabile `lPtr` come puntatore a un oggetto di tipo `long`.
b) Assegnate l'indirizzo della variabile `value1` al puntatore `lPtr`.
c) Visualizzate il valore dell'oggetto a cui punta `lPtr`.
d) Assegnate il valore dell'oggetto a cui punta `lPtr` alla variabile `value2`.
e) Visualizzate il valore di `value2`.
f) Visualizzate l'indirizzo di `value1`.
g) Visualizzate l'indirizzo contenuto in `lPtr`. Il valore visualizzato è uguale all'indirizzo di `value1`?

5.11 Eseguite le seguenti operazioni:

a) Scrivete l'intestazione della funzione `zero` che riceve come parametro l'array di interi lunghi `bigIntegers` e non restituisce alcun valore.
b) Scrivete il prototipo di funzione di `bigIntegers`.
c) Scrivete l'intestazione della funzione `add1AndSum` che prende come parametro l'array di interi `oneTooSmall` e restituisce un valore intero.

Scrivete il prototipo di funzione di `add1AndSum`.

[Nota: Gli esercizi dal 5.12 al 5.15 sono un po' più difficili. Se riuscite ad affrontare questi problemi non dovreste incontrare difficoltà nell'implementare i giochi di carte più comuni.]

5.12 Modificate il programma in Figura 5.24 in modo che la funzione che distribuisce la carte gestisca una mano di poker (5 carte). Riscrivete la funzione in modo tale che dopo aver distribuito le carte per la mano di poker riesca a:

a) Determinare se c'è una coppia.
b) Determinare se c'è una doppia coppia.
c) Determinare se c'è un tris di un certo tipo (per es. un tris di fanti).
d) Determinare se c'è un poker di un certo tipo (per es. quattro assi).
e) Determinare se c'è un colore (cioè quattro carte dello stesso seme).
f) Determinare se c'è una scala reale (cioè cinque carte dai valori consecutivi).

5.13 Utilizzate la funzione sviluppata nell'esercizio 5.12 per scrivere un programma che distribuisca le carte per due mani di poker (5 carte per giocatore), confronti le due mani e determini qual è la migliore.

5.14 Modificate il programma sviluppato nell'esercizio 5.13 in modo da simulare un giocatore. Le cinque carte sono tenute con la figura rivolta verso il basso, in modo che il giocatore umano non possa vederle. Il programma dovrebbe valutare la propria mano e, sulla base della sua qualità, dovrebbe chiedere al giocatore umano se rimpiazzare una o più carte (prendendone di nuove dal mazzo) che non servivano nella mano originale. Il programma dovrebbe poi valutare nuovamente la propria mano (Attenzione: qui le cose si complicano!).

5.15 Modificate il programma sviluppato nell'esercizio 5.14 in modo che possa gestire automaticamente la propria mano e consenta al giocatore di cambiare le proprie carte. Il programma dovrebbe poi confrontare le due mani per determinare chi vince. In un momento successivo utilizzate questo programma per 20 giocate contro il computer. Chi vince di più, voi o il computer? Fateci giocare un vostro amico per 20 volte. Chi vince di più? Sulla base dei risultati di queste giocate modificate il vostro programma per ridefinire le sue giocate (anche questo è un esercizio difficile). Fate altre 20 giocate. Com'è il gioco del computer? Migliore di prima?

5.16 Nel programma in Figura 5.24 abbiamo utilizzato intenzionalmente un algoritmo di mescolamento inefficiente, che poneva il problema del differimento indefinito. In questo esercizio creerete un algoritmo molto più efficiente, che non pone più quel problema.
Modificate il programma in Figura 5.24 come segue: inizializzate l'array `deck` come in Figura 5.38 e modificate la funzione `shuffle` in modo che consideri ogni elemento dell'array una sola volta, scandendo riga per riga e colonna per colonna. Ogni elemento dovrebbe essere scambiato con un altro elemento dell'array scelto a caso. Visualizzate l'array risultante, per capire se `deck` è stato mescolato a sufficienza (come in Figura 5.36, per esempio). Magari potreste desiderare che `shuffle` sia chiamata diverse volte, per mescolare meglio il mazzo. Notate che stiamo affrontando soltanto il problema del mescolamento: l'algoritmo di distribuzione delle carte ha ancora bisogno di ricercare in tutto l'array la carta 1, poi la carta 2 e così via. Peggio ancora, anche dopo aver localizzato le carte nell'ordine, l'algoritmo continua la ricerca nel resto di `deck`. Modificate il programma in Figura 5.24 in modo che una volta che una carta sia stata estratta dal mazzo, non venga effettuato più alcun tentativo di trovare quel numero d'ordine. Il programma, insomma, dovrebbe proseguire direttamente a ricercare la carta successiva.

L'array deck prima del mescolamento

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Figura 5.38 L'array deck prima del mescolamento.

L'array deck dopo un mescolamento

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Figura 5.39 L'array deck dopo un mescolamento.

5.17 (*Simulazione: la tartaruga e la lepre*) In questo problema ricreerete la famosa gara fra la tartaruga e la lepre. Utilizzerete i numeri casuali per la simulazione di questo evento memorabile.
I nostri concorrenti iniziano la gara dal "quadrato 1", il primo di 70 quadrati disegnati sull'asfalto. Ogni quadrato è una posizione possibile lungo il percorso della gara. La bandierina finale si trova sul quadrato 70. Il primo concorrente che raggiunge o sorpassa il quadrato 70 vince un succulento cestino pieno di carote e lattuga. Il percorso si snoda lungo il versante in salita di una collinetta, per cui capita che uno dei due concorrenti possa perdere un po' di terreno. Un timer incrementa il suo valore di 1 tic al secondo. Ad ogni tic del timer il programma aggiorna la posizione degli animali, secondo le seguenti regole.

Animale	Mossa	% di tempo	Spostamento
Tartaruga	Passo veloce	50%	3 quadrati a destra
	Scivolata	20%	6 quadrati a sinistra
	Passo lento	30%	1 quadrato a destra
Lepre	Riposo	20%	Nessuno spostamento
	Salto lungo	20%	9 quadrati a destra
	Scivolata lunga	10%	12 quadrati a sinistra
	Salto piccolo	30%	1 quadrato a destra
	Scivolata piccola	20%	2 quadrati a sinistra

Utilizzate delle variabili per tener traccia delle posizioni degli animali (ovvero del quadrato su cui si trovano, nella gamma 1-70). Fate partire ogni animale dalla posizione 1. Se un animale scivola in una posizione precedente al quadrato 1, abbiate pietà e rimettetelo nuovamente sul quadrato 1.
Generate le percentuali della tabella precedente con un numero intero casuale i nella gamma 1 ≤ i ≤ 10. Per la tartaruga, fatela andare a passo veloce quando 1 ≤ i ≤ 5, fatela scivolare quando 6 ≤ i ≤ 7, oppure fatela andare a passo lento quando 8 ≤ i ≤ 10. Usate una tecnica simile per spostare la lepre.
Ogni gara che si rispetti comincia con lo sparo iniziale. Visualizzate

```
BANG !!!!!
PARTITI !!!!!
```

Ad ogni tic del timer (ovvero a ogni iterazione del ciclo), visualizzate una linea con 70 tacche, dove mostrate la posizione della tartaruga con una `T` e quella della lepre con una `L`. Può anche capitare che i due concorrenti capitino sullo stesso quadrato. In questo caso la tartaruga morde la lepre e il vostro programma scriverà un `OUCH!!!` in quella posizione. Tutte gli spazi di visualizzazione dove non c'è ne la `T`, né la `L` e neppure `OUCH!!!` dovrebbero restare vuoti.

Dopo aver visualizzato ogni linea, controllate se uno degli animali ha raggiunto o sorpassato il quadrato 70. Se è così, visualizzate il nome del vincitore e terminate la simulazione. Se vince la tartaruga visualizzate `HA VINTO LA TARTARUGA!!! YAHOO!!!` Per par condicio, utilizzate una scritta del genere anche se vince la lepre, naturalmente. Diamine! E se entrambi raggiungono il quadrato 70 nello stesso momento? Fate come credete: se volete potete favorire in ogni caso la tartaruga oppure declamare una rivincita. Se nessuno degli animali vince, effettuate il ciclo di nuovo per il successivo tic del timer. Quando il vostro programma sarà pronto radunate un gruppo di tifosi. Non crederete ai vostri occhi quando vedrete come si infervorerà la platea!

Sezione speciale: costruite il vostro computer

Nei prossimi esercizi siamo intenzionati a esaminare la struttura interna di un computer aprendo l'unità ed analizzandone ogni suo componente. Introduciamo un po' di programmazione in linguaggio macchina, scrivendo anche alcuni programmini. Per rendere questa esperienza ancora più importante proviamo a costruire un computer grazie alle tecniche di simulazione software: provate a far girare i vostri programmini in linguaggio macchina su questo computer!

5.18 (*Programmazione in linguaggio macchina*) Cominciamo a creare il nostro computer, lo chiameremo Simpletron. Come suggerisce il suo nome, si tratta di una macchina piuttosto semplice, ma vedremo anche abbastanza potente. Simpletron esegue solo programmi scritti nell'unico linguaggio che capisce direttamente, cioè il linguaggio macchina, che abbreviamo in SML (Simpletron Machine Language).
Simpletron contiene un *accumulatore*, che è un registro speciale in cui vengono messi i dati prima che Simpletron li esamini o li usi per effettuare calcoli di vario tipo. Tutti i dati di Simpletron sono trattati in termini di *parole*. Una parola è un numero decimale a quattro cifre, come +3364, -1293, +0007, -0001 e così via. Simpletron ha una capacità di memoria di 100 parole, e le locazioni di memoria vengono riferite con i numeri d'ordine 00, 01, ..., 99.
Prima di eseguire un programma in SML, occorre caricarlo in memoria; per convenzione, la prima istruzione di ogni programma in SML viene posta nell'indirizzo 00 e il simulatore avvierà sempre *l'esecuzione a partire da questa locazione. Ogni istruzione scritta in SML occupa una parola della* memoria di Simpletron e, dunque, possiamo pensarle come numeri decimali con segno a quattro cifre. Supporremo che il segno di un'istruzione in SML sia sempre positivo, mentre il segno delle parole di dati può essere positivo o negativo. Un indirizzo di memoria di Simpletron può contenere un'istruzione, un dato utilizzato da un programma o una zona di memoria non utilizzata (e non definita). Le prime due cifre di ogni istruzione in SML costituiscono il *codice operativo*, e specificano l'operazione da eseguire; i codici operativi di SML sono elencati in Figura 5.40

Codice dell'operazione	Significato
Operazioni di input/output:	
`const int READ = 10`	Legge una parola dalla tastiera e la trasferisce in un indirizzo di memoria specificato.
`const int WRITE = 11;`	Legge una parola da un indirizzo di memoria specifico e la visualizza sullo schermo.
Operazioni di caricamento/memorizzazione:	
`const int LOAD = 20;`	Carica nell'accumulatore una parola presente in un indirizzo di memoria specifico.
`const int STORE = 21;`	Memorizza in un indirizzo di memoria specifico la parola contenuta nell'accumulatore.

Figura 5.40 Codici operativi in SML (continua)

Codice dell'operazione	Significato
Operazioni aritmetiche:	
`const int ADD = 30;`	Addiziona una parola presente in un indirizzo di memoria specifico alla parola contenuta nell'accumulatore (il risultato resta nell'accumulatore).
`const int SUBTRACT = 31;`	Sottrae una parola presente in un indirizzo di memoria specifico dalla parola contenuta nell'accumulatore (il risultato resta nell'accumulatore).
`const int DIVIDE = 32;`	Divide una parola presente in un indirizzo di memoria specifico per la parola contenuta nell'accumulatore (il risultato resta nell'accumulatore).
`const int MULTIPLY = 33;`	Moltiplica una parola presente in un indirizzo di memoria specifico per la parola contenuta nell'accumulatore (il risultato resta nell'accumulatore).
Operazioni per il trasferimento del controllo:	
`const int BRANCH = 40;`	Saltare all'indirizzo di memoria specificato.
`const int BRANCHNEG = 41;`	Saltare all'indirizzo di memoria specificato se l'accumulatore è negativo.
`const int BRANCHZERO = 42;`	Saltare all'indirizzo di memoria specificato se l'accumulator vale zero.
`const int HALT = 43;`	Alt! Il programma è terminato.

Figura 5.40 Codici operativi in SML.

Le ultime due cifre di un'istruzione in SML sono l'*operando*, cioè l'indirizzo di memoria che contiene la parola con cui effettuare l'operazione.

Prendiamo in rassegna alcuni semplici programmini in SML. Il primo (Esempio 1) legge due numeri dalla tastiera, ne calcola la somma e la visualizza. L'istruzione `+1007` legge il primo numero dalla tastiera e lo trascrive nell'indirizzo `07` (che è stata inizializzata a zero). La successiva istruzione `+1008` legge il numero successivo e lo trascrive nell'indirizzo `08`. L'istruzione di caricamento `+2007`, mette (copia) il primo numero nell'accumulatore e l'istruzione `+3008` addiziona il secondo numero al numero che si trova nell'accumulatore. Tutte le istruzioni aritmetiche in SML lasciano il risultanto nell'accumulatore. L'istruzione di memorizzazione `+2109` pone (copia) il risultato nuovamente in memoria, nell'indirizzo `09`. Sarà lì che andrà a leggere l'istruzione di visualizzazione `+1109` per visualizzare il valore della somma come numero decimale con segno a quattro cifre. L'istruzione di Alt! `+4300` termina l'esecuzione.

Esempio 1

Locazione	Numero	Istruzione
`00`	`+1007`	`(Leggi A)`
`01`	`+1008`	`(Leggi B)`
`02`	`+2007`	`(Carica A)`
`03`	`+3008`	`(Addiziona B)`
`04`	`+2109`	`(Memorizza C)`
`05`	`+1109`	`(Visualizza C)`

(continua)

Esempio 1

Locazione	Numero	Istruzione
`06`	`+4300`	`(Alt)`
`07`	`+0000`	`(Variabile A)`
`08`	`+0000`	`(Variabile B)`
`09`	`+0000`	`(Risultato C)`

Il programma in SML dell'Esempio 2 legge due numeri dalla tastiera, ne determina il maggiore e lo visualizza. Notate l'uso dell'istruzione `+4107` come istruzione condizionale per il trasferimento del controllo, simile a un costrutto `if` del C++.

Esempio 2

Locazione	Numero	Istruzione
`00`	`+1009`	`(Leggi A)`
`01`	`+1010`	`(leggi B)`
`02`	`+2009`	`(Carica A)`
`03`	`+3110`	`(Sottrai B)`
`04`	`+4107`	`(Se negativo, vai a 07)`
`05`	`+1109`	`(Visualizza A)`
`06`	`+4300`	`(Alt)`
`07`	`+1110`	`(Visualizza B)`
`08`	`+4300`	`(Alt)`
`09`	`+0000`	`(Variabile A)`
`10`	`+0000`	`(Variabile B)`

Adesso provate a scrivere dei programmi in SML che eseguano i seguenti compiti:

a) Utilizzate un ciclo controllato da un valore sentinella per leggere 10 valori positivi, e calcolate e visualizzate la loro somma.
b) Utilizzate un ciclo controllato da un contatore per leggere 7 numeri, alcuni positivi e altri negativi, e calcolate e visualizzate la loro media.
c) Leggete una serie di numeri per determinare e visualizzare il valore maggiore. Il primo numero letto indica quanti sono i numeri da elaborare.

5.19 (*Simulatore di computer*) Potrà sembrarvi perlomeno coraggioso, ma vi assicuriamo che in questo esercizio costruirete un computer tutto vostro. Beh, no, non utilizzando dei circuiti integrati ed un saldatore; piuttosto utilizzerete le tecniche di simulazione per creare un modello software di Simpletron. Non resterete delusi: il vostro simulatore di Simpletron trasformerà il vostro computer in un vero e proprio Simpletron, per cui sarete in grado di eseguire e testare i programmi in SML dell'esercizio 5.18. Quando lanciate il simulatore di Simpletron, dovrebbe apparire per prima cosa:

```
*** Benvenuti in Simpletron! ***
*** Digitate il vostro programma un'istruzione ***
*** (o una parola di dati) alla volta. Io visualizzerà il numero ***
*** dell'indirizzo di memoria e un punto interrogativo (?). ***
*** Quindi voi dovrete digitare la parola per tale locazione. ***
*** Digitate il valore sentinella -99999 per terminare ***
*** il vostro programma. ***
```

Simulate la memoria di Simpletron con un array unidimensionale di 100 elementi. Supponiamo che il simulatore sia ora in esecuzione, ed esaminiamo l'interazione con il programma man man che digitiamo l'Esempio 2 dell'esercizio 5.18:

```
00 ?.+1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Caricamento del programma completato ***
*** Inizio dell'esecuzione ***
```

Il programma in SML è stato caricato nell'array, ovvero nella memoria di Simpletron, ed ora il compito di Simpletron è di eseguirlo. L'esecuzione comincia dall'indirizzo `00` e, come in C++, continua sequenzialmente, a meno che non ci siano istruzioni che ne trasferiscono il controllo altrove.
Utilizzate la variabile `accumulator` per rappresentare il registro accumulatore. La variabile `counter` tiene traccia dell'indirizzo che contiene l'istruzione da eseguire. La variabile `operationCode` indica l'operazione in corso, cioè le due cifre di sinistra della parola dell'istruzione. La variabile `operand` indica l'indirizzo di memoria su cui opera l'istruzione corrente. Quindi `operand` corrisponde alle ultime due cifre dell'istruzione in corso. Non eseguite le istruzioni direttamente dalla memoria. Invece trasferite l'istruzione successiva dalla memoria in una variabile di nome `instructionRegister`. Dopo di che prendete le due cifre di sinistra e ponetele in `operationCode`, mentre le cifre di destra andranno in `operand`. Quando Simpletron avvia l'esecuzione, i registri speciali sono tutti inizializzati a 0
Adesso seguiamo passo passo l'esecuzione della prima istruzione, `+1009`, che si trova nell'indirizzo di memoria 00. Questo procedimento si chiama *ciclo di esecuzione di un'istruzione*.
`counter` contiene l'indirizzo della istruzione successiva. Prendiamo il contenuto di tale indirizzo di memoria utilizzando l'istruzione C++

```
instructionRegister = memory[ counter ];
```

Le seguenti istruzioni estraggono il codice operativo e l'operando dal registro delle istruzioni

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Adesso Simpletron deve determinare di che codice operativo si tratta. Un costrutto `switch` differenzia le dodici operazioni di SML.

Le etichette del costrutto `switch` simulano il comportamento delle diverse istruzioni SML in questo modo (lasciamo le altre al lettore):

read:	`cin >> memory[ operand ];`
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
branch:	Parleremo tra breve di questa istruzione.
halt:	Visualizza il messaggio `*** Esecuzione di Simpletron terminata ***`

e visualizza (i) il nome e il contenuto di ogni registro e (ii) il contenuto di tutta la memoria. La visualizzazione di un'intera area di memoria viene anche detta *dump* (in inglese discarica... ma non ci riferiamo al luogo dove vanno i vecchi computer). Per comprendere come scrivere la vostra funzione di dump osservate il formato di un dump in Figura 5.4.


```
REGISTRI:
accumulator    +0000
counter        00
instructionRegister  +0000
operationCode  00
operand        00

MEMORIA:
         0     1     2     3     4     5     6     7     8     9
  0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Figura 5.41 Esempio di un dump di memoria.

Se effettuate un dump dopo l'esecuzione di un programma di Simpletron, vedrete le istruzioni e il valore di dati al termine dell'esecuzione. Andiamo avanti con l'esecuzione della prima istruzione del nostro programma: `+1009` nell'indirizzo 00. Come abbiamo indicato, l'istruzione `switch` la simula eseguendo l'istruzione C++

```
cin >> memory[ operand ];
```

Sullo schermo dovrebbe comparire un punto interrogativo (?) prima dell'esecuzione di `cin`, per avvertire l'utente che il programma attende un input. Simpletron aspetta che l'utente immetta un valore e prema il tasto Invio. Il valore viene poi letto e trasferito nell'indirizzo di memoria `09`.
A questo punto la simulazione della prima istruzione è completa. Tutto ciò che resta da fare è preparare Simpletron a eseguire l'istruzione successiva. Dato che l'ultima istruzione non era un trasferimento del controllo, dobbiamo semplicemente incrementare il registro `counter`:

```
++counter;
```

Adesso la simulazione della prima istruzione è davvero completa. L'intero procedimento (il ciclo di esecuzione) comincia da capo per la successiva istruzione.
Adesso vediamo un po' come sono eseguite le istruzioni di salto. Dobbiamo essenzialmente regolare il valore del registro `counter` nel modo corretto. Un'istruzione di salto può essere simulata con

```
counter = operand;
```

Il salto condizionale (se l'accumulatore vale zero) è simulata invece da

```
if ( accumulator == 0 )
   counter = operand;
```

A questo punto dovreste essere in grado di implementare il vostro simulatore Simpletron e di eseguire i programmi che avete scritto nell'esercizio 5.18. Potete anche abbellire ed estendere il linguaggio SML a vostro piacimento.
Il vostro simulatore dovrebbe essere in grado di rilevare vari tipi di errori nei programmi SML. Per esempio, durante il caricamento del programma dovrebbe controllare che i numeri in memoria siano compresi nell'intervallo da `-9999` a `+9999`. Utilizzate un ciclo `while` per verificare che ogni numero immesso si trovi in questo intervallo, e se non è così chiedete nuovamente all'utente di immettere un numero valido. Durante l'esecuzione, il simulatore dovrebbe anche essere in grado di accorgersi di errori più seri, come il tentativo di dividere per zero, l'esecuzione di codici operativi inesistenti oppure

un eventuale overflow dell'accumulatore (un'operazione aritmetica che dà risultati fuori dall'intervallo tra –9999 e +9999). Errori di questo tipo si chiamano *errori fatali*. Se il simulatore rileva un errore fatale dovrebbe scrivere un messaggio d'errore del tipo

```
*** Tentativo di dividere per zero ***
*** Esecuzione di Simpletron terminata a causa di un errore ***
```

e dovrebbe visualizzare un dump completo nel formato che abbiamo già visto. In questo modo sarà più facile localizzare l'errore nel programma.

Ulteriori esercizi sui puntatori

5.20 Modificate il programma delle carte di Figura 5.24 in modo che sia la stessa funzione a mescolare e distribuire le carte (chiamamola `shuffleAndDeal`). Questa funzione dovrebbe contenere un ciclo nidificato simile a quello della funzione `shuffle` in Figura 5.24.

5.21 Che cosa fa questo programma?

```
1   // ex05_21.cpp
2   #include <iostream.h>
3
4   void mystery1( char *, const char * );
5
6   int main()
7   {
8      char string1[ 80 ], string2[ 80 ];
9
10     cout << "Enter two strings: ";
11     cin >> string1 >> string2;
12     mystery1( string1, string2 );
13     cout << string1 << endl;
14     return 0;
15  }
16
17  void mystery1( char *s1, const char *s2 )
18  {
19     while ( *s1 != '\0' )
20        ++s1;
21
22     for ( ; *s1 = *s2; s1++, s2++ )
23        ;  // empty statement
24  }
```

5.22 Che cosa fa questo programma?

```
1   // ex05_22.cpp
2   #include <iostream.h>
3
4   int mystery2( const char * );
5
6   int main()
7   {
8      char string[ 80 ];
9
10     cout << "Enter a string: ";
11     cin >> string;
12     cout << mystery2( string ) << endl;
13     return 0;
14  }
15
```



```
16  int mystery2( const char *s )
17  {
18     for ( int x = 0; *s != '\0'; s++ )
19        ++x;
20
21     return x;
22  }
```

5.23 *Trovate l'errore in ognuno di questi segmenti di programma. Se è possibile correggerlo, spiegate come.*

a)
```
int *number;
cout << number << endl;
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
for ( ; *s != '\0'; s++)
   cout << *s << ' ';
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
float x = 19.34;
float xPtr = &x;
cout << xPtr << endl;
```

g)
```
char *s;
cout << s << endl;
```

5.24 (*Quicksort*) Negli esempi e negli esercizi del Capitolo 4 abbiamo parlato delle tecniche di ordinamento bubble sort, bucket sort e selection sort. Qui presentiamo una tecnica ricorsiva nota come Quicksort (in inglese *ordinamento veloce*). L'algoritmo di base per un array unidimensionale è il seguente:

a) *Partizionamento*: Prendete il primo elemento di un array non ordinato e determinate il suo indirizzo finale nell'array ordinato, vale a dire tutti i valori a sinistra dell'elemento sono minori e tutti i valori a destra sono maggiori. A questo punto si ha un elemento nell'indirizzo giusto e due sottoarray non ordinati.

b) *Passo ricorsivo*: Effettuare l'operazione precedente su ciascun sottoarray.

Ogni volta che si effettua il partizionamento su di un sottoarray si ottiene un elemento posizionato nell'indirizzo corretto e una coppia di sottoarray più piccoli. Quando si ottiene un sottoarray costituito da un solo elemento, esso è banalmente ordinato.

L'algoritmo di base sembra piuttosto semplice, ma il problema è come determinare la posizione finale del primo elemento di ciascun sottoarray. Prendiamo come esempio questo insieme di valori (l'elemento in grassetto è quello del partizionamento, cioè sarà posto nell'indirizzo finale nell'array ordinato):

37 2 6 4 89 8 10 12 68 45

a) Partendo dall'ultimo elemento a destra confrontiamo ciascun elemento con 37 finché non ne troviamo uno minore di 37. Se accade bisogna scambiare di posto 37 e quell'elemento. Il primo elemento minore di 37 è 12, per cui i due valori si scambiano di posto. Il nuovo array è:

12 2 6 4 89 8 10 37 68 45

L'elemento 12 è in corsivo per indicare che è stato appena scambiato di posto con 37.

b) Partendo dalla sinistra dell'array, dall'elemento che si trova subito dopo 12, confrontiamo ogni elemento con 37 finché non ne troviamo uno maggiore di 37. Se accade, scambiamo di posto 37 e quell'elemento. Il primo elemento maggiore che incontriamo è 89, per cui lo scambiamo di posto con 37. Il nuovo array è:

   12 2 6 4 37 8 10 *89* 68 45

c) Ripartiamo da destra, ma dall'elemento che si trova prima di 89, confrontiamo ogni elemento con 37, alla ricerca di un altro valore minore di 37, che scambieremo di posto. Il primo elemento minore di 37 è 10. Effettuiamo lo scambio, creando il nuovo array:
   12 2 6 4 *10* 8 37 89 68 45

d) Ripartiamo da sinistra, ma dall'elemento che si trova dopo 10, cercando il prossimo elemento maggiore di 37. Tuttavia non ne troviamo nessuno, fino allo stesso 37, per cui sappiamo che 37 si trova nella posizione finale dell'array ordinato.

Adesso che abbiamo partizionato l'array originario, abbiamo due sottoarray non ordinati. Il sottoarray con i valori minori di 37 contiene 12, 2, 6, 4, 10 e 8. Il sottoarray con i valori maggiori di 37 contiene 89, 68 e 45. L'ordinamento continua in entrambi i sottoarray, nello stesso modo che abbiamo visto per l'array originario.

Sulla base di questa discussione, scrivete la funzione ricorsiva `quickSort` che ordina un array di interi unidimensionale. La funzione dovrebbe ricevere come argomenti l'array di interi, l'indice iniziale e l'indice finale. La funzione `quickSort` dovrebbe chiamare la funzione `partition` per effettuare il partizionamento.

5.25 (*Uscita da un labirinto*) Questa griglia è formata da # e puntini (.), ed è la rappresentazione di un labirinto tramite un array bidimensionale.

```
# # # # # # # # # # # #
#   .   #   .   .   .   #
#   .   #   .   #   #   #
#   .   .   .   #   #   #
# # # #   #   #   .   #
#   .   #   #   #   .   #
#   .   #   #   #   .   #
# #   #   #   #   #   .   #
#   .   .   .   .   #   .   #
# # # # # #   # # #   .   #
#   .   .   .   #   .   .   #
# # # # # # # # # # # # #
```

I segni # rappresentano i muri del labirinto, mentre i puntini rappresentano i percorsi possibili al suo interno. Potete spostarvi soltanto sulle locazioni dell'array che contengono un puntino.

Esiste un semplice algoritmo per attraversare un labirinto e trovarne sicuramente l'uscita (se c'è). Se non c'è un'uscita si torna nuovamente nell'indirizzo di partenza. Mettete la vostra mano destra sul muro alla vostra destra e cominciate a camminare in avanti. Non allontanate mai la mano dal muro. Se il labirinto gira a destra lo farete anche voi. Se non togliete mai la mano dal muro riuscirete ad arrivare all'uscita. È probabile che ci siano delle scorciatoie o dei percorsi un po' più brevi, ma seguendo questo algoritmo avrete la certezza di guadagnare l'uscita. Scrivete la funzione ricorsiva `mazeTraverse` per uscire dal labirinto. La funzione riceverà come argomento un array di caratteri 12x12, che rappresenta il labirinto, e l'indirizzo iniziale da cui partite. Man man che `mazeTraverse` cerca di guadagnare l'uscita dal labirinto, dovrebbe provvedere a segnalare con una `X` i percorsi che batte. La funzione dovrebbe poi aggiornare la visualizzazione del labirinto ad ogni mossa, in modo che l'utente possa rendersi conto del percorso scelto dal programma per uscire.

5.26 (*Generazione casuale di labirinti*) Scrivete la funzione `mazeGenerator` che riceve come argomento un array di caratteri bidimensionale 12x12 e produce un labirinto casuale. La funzione deve prevedere un punto iniziale e finale nel labirinto. Mettete alla prova la funzione `mazeTraverse` dell'Esercizio 5.25 facendole attraversare diversi labirinti casuali.

5.27 (*Labirinti di ogni dimensione*) Generalizzate le funzioni `mazeTraverse` e `mazeGenerator` degli Esercizi 5.25 e 5.26 per labirinti di qualsiasi larghezza e altezza.

5.28 (*Array di puntatori a funzioni*) Riscrivete il programma in Figura 4.23 in modo che utilizzi un'interfaccia a menu. Il programma dovrebbe consentire all'utente di scegliere una tra 5 opzioni, come segue (visualizzatele sullo schermo!):

```
Digitate la vostra scelta:
  0 Visualizzare l'array di voti
  1 Trovare il voto più basso
  2 Trovare il voto più alto
  3 Visualizzare la media dei test per ogni studente
  4 Terminare il programma
```

Una restrizione sull'uso degli array di puntatori è che tutti i puntatori devono essere dello stesso tipo. Perciò i puntatori devono puntare a funzioni che restituiscono lo stesso tipo e prendono argomenti dello stesso tipo. Per questo motivo dobbiamo uniformare le funzioni in Figura 4.23. Modificate le funzioni `minimum` e `maximum` in modo che visualizzino il valore minimo e massimo e non restituiscano alcun dato. Per l'opzione 3, modificate la funzione `average` in Figura 4.23 in modo che visualizzi la media di ogni studente, e non di uno studente specifico. Anche `average` non deve restituire alcun dato e prendere gli stessi parametri di `printArray`, `minimum` e `maximum`. Memorizzate i puntatori alle quattro funzioni nell'array `processGrades` e utilizzate la scelta dell'utente come indice dell'array per chiamare ciascuna funzione.

5.29 (*Modifiche al simulatore di Simpletron*) Nell'Esercizio 5.19 avete scritto un software che simula un computer ed esegue programmi scritti in SML. Qui vi proponiamo di migliorare e modificare il simulatore di Simpletron in diversi punti. Negli Esercizi 15.26 e 15.27 vi proponiamo di scrivere un compilatore che traduca i programmi scritti in un linguaggio ad alto livello (una variante del BASIC) in SML. Per eseguire i programmi prodotti dal compilatore crediamo sia meglio apportare alcune modifiche al simulatore.

a) Estendete la memoria del simulatore di Simpletron a 1000 locazioni, per eseguire programmi più grandi.
b) Inserite nelle specifiche del linguaggio SML l'istruzione che effettua l'operazione modulo.
c) Inserite nelle specifiche del linguaggio SML l'istruzione esponente.
d) Modificate il simulatore in modo che rappresenti le istruzioni in SML con valori esadecimali anziché decimali.
e) Modificate il simulatore in modo che sia in grado di visualizzare un carattere di nuova linea. Dovrete aggiugere un'altra istruzione SML.
f) Modificate il simulatore in modo che possa elaborare valori in virgola mobile oltre ai valori interi.
g) Modificate il simulatore in modo che sappia gestire l'input di stringhe. Suggerimento: Ogni parola di Simpletron può essere divisa in due segmenti, ognuno formato da due cifre intere. Un intero a due cifre rappresenta anche un carattere, secondo il set ASCII. Aggiungete un'istruzione in linguaggio macchine che riceve in input una stringa e memorizza la stringa a partire da uno specifico indirizzo di memoria di Simpletron. La prima metà della parola di quell'indirizzo corrisponderà al numero di caratteri della stringa, ovvero alla sua lunghezza. Tutte le semiparole che seguono conterranno un carattere ASCII, espresso appunto nella forma di un numero a due cifre. Un'istruzione in linguaggio macchina converte ogni carattere nel suo equivalente ASCII e lo assegna a una semiparola.

h) Modificate il simulatore in modo che sappia gestire l'output delle stringhe memorizzate secondo il formato del punto (g). Suggerimento: Aggiungete un'istruzione in linguaggio macchina che visualizza una stringa che inizia in uno specifico indirizzo di memoria di Simpletron. La prima semiparola di quell'indirizzo contiene la lunghezza della stringa. Le semiparole che seguono contengono ognuna un carattere ASCII espresso nella forma di un numero di due cifre. L'istruzione in linguaggio macchina controlla la lunghezza e visualizza la stringa convertendo ogni numero a due cifre nel carattere corrispondente.

5.30 Che cosa fa questo programma?

```
1  // ex05_30.cpp
2  #include <iostream.h>
3
4  int mystery3( const char *, const char * );
5
6  int main()
7  {
8     char string1[ 80 ], string2[ 80 ];
9
10    cout << "Enter two strings: ";
11    cin >> string1 >> string2;
12    cout << "The result is "
13         << mystery3( string1, string2 ) << endl;
14
15    return 0;
16 }
17
18 int mystery3( const char *s1, const char *s2 )
19 {
20    for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )
21
22       if ( *s1 != *s2 )
23          return 0;
24
25    return 1;
26 }
```

Esercizi sulla manipolazione di stringhe

5.31 Scrivete un programma che utilizza la funzione `strcmp` per confrontare due stringhe immesse dall'utente. Il programma deve essere in grado di determinare se la prima stringa è uguale, minore o maggiore della seconda.

5.32 Scrivete un programma che utilizza la funzione `strncmp` per confrontare due stringhe immesse dall'utente. Il programma deve chiedere all'utente anche il numero di caratteri su cui effettuare il confronto. Il programma deve poi determinare se la prima stringa è uguale, minore o maggiore della seconda

5.33 Scrivete un programma che utilizza i numeri casuali per formare delle frasi in inglese. Il programma utilizzerà quattro array di puntatori a `char` di nome `articolo`, `nome`, `verbo` e `preposizione`. Il programma deve creare una frase selezionando una parola a caso da ciascun array, in quest'ordine: `articolo`, `nome`, `verbo`, `preposizione`, `articolo` e `nome`. Man mano che viene pescata una nuova parola, questa deve essere concatenata alle parole già scelte, in modo da costruire una stringa che contiene la frase. Separate le parola con gli spazi. Quando visualizzate la frase generata, ponete la prima lettera in maiuscolo e terminate la visualizzazione con un punto. Fate generare al programma 20 frasi.

Gli array contengono le seguenti parole: l'array `articolo` contiene `"the"`, `"a"`, `"one"`, `"some"` e `"any"`; l'array `nome` contiene i nomi `"boy"`, `"girl"`, `"dog"`, `"town"` e `"car"`; l'array `verbo` contiene i verbi `"drove"`, `"jumped"`, `"ran"` `"walked"` e `"skipped"`; l'array `preposizione` contiene le preposizioni `"to"`, `"from"`, `"over"`, `"under"` e `"on"`
Dopo aver scritto questo programma, modificatelo per produrre una storiella, formata da diverse frasi di questo tipo (mica male fare lo "scrittore casuale", eh?)

5.34 (*Limerick*) Un limerick è una composizione in versi di cinque righe in cui la prima e la seconda riga sono in rima con la quinta, e la terza è in rima con la quarta. Con tecniche come quelle che abbiamo sviluppato nell'Esercizio 5.33, scrivete un programma in C++ che genera limerick a caso. Affinare il gusto del programma, per generare limerick degni di essere letti, sarà anche un problema difficile, ma pensate al risultato!

5.35 Scrivete un programma che codifica le frasi inglesi nel cosiddetto *pig Latin*. Il pig Latin è una forma di linguaggio spesso utilizzata dai ragazzini per puro divertimento. Esistono, tra l'altro, molte varianti di questo strano linguaggio. Per semplicità utilizziamo questo algoritmo:
Per formare una frase in pig Latin da una frase inglese, estraete le singole parole della frase (sono dei token) con la funzione `strtok`. Per tradurre ogni parola inglese in pig Latin, scrivete la prima lettera della parola alla fine e aggiungetele le lettere `"ay"`. Così facendo, dalla parola `"jump"` avremo `"umpjay"`, e dalla parola `"the"` avremo `"hetay"` Gli spazi tra le parole rimangono anche nella frase finale. Facciamo questa assunzione: la frase inglese sarà formata da parole separate da spazi bianchi, senza segni di punteggiatura, e tutte le parole sono almeno di due lettere. La funzione `printLatinWord` è deputata a visualizzare ogni parola. Suggerimento: Ad ogni token che trovate con `strtok`, passate il puntatore al token alla funzione `printLatinWord` e visualizzate la parola in pig Latin.

5.36 Scrivete un programma che riceve in input un numero di telefono, come una stringa nella forma `(555) 555-5555`. Il programma dovrebbe utilizzare la funzione `strtok` per estrarre il prefisso come primo token, le prime tre cifre del numero come secondo token e le quattro cifre di coda del numero come ultimo token. Le sette cifre del numero di telefono devono essere concatenate in una stringa. Il programma deve infine convertire il prefisso in un `int` e il numero in un `long`. Visualizzate i due numeri risultanti.

5.37 Scrivete un programma che riceve in input una linea di testo, estrae i token contenuti in essa e li visualizza in ordine inverso.

5.38 Utilizzate le funzioni di confronto tra stringhe (Sezione 5.12.2) e le tecniche di ordinamento degli array per scrivere un programma che pone in ordine alfabetico un elenco di stringhe. Utilizzate i nomi di 10 o 15 città per verificare il funzionamento del programma.

5.39 Scrivete due versioni delle funzioni di copia e concatenamento di stringhe in Figura 5.29. Nella prima versione utilizzate l'indicizzazione dell'array e nella seconda i puntatori e l'aritmetica dei puntatori.

5.40 Scrivete due versioni della funzione di confronto tra stringhe in Figura 5.29. Nella prima versione utilizzate l'indicizzazione dell'array e nella seconda i puntatori e l'aritmetica dei puntatori.

5.41 Scrivete due versioni della funzione `strlen` in Figura 5.29. Nella prima versione utilizzate l'indicizzazione dell'array e nella seconda i puntatori e l'aritmetica dei puntatori.

Sezione speciale: esercizi avanzati sulla manipolazione di stringhe

Gli esercizi precedenti si concentrano sui concetti fondamentali del trattamento dei testi e delle stringhe. Questa sezione contiene degli esercizi di livello intermedio e avanzato. Per risolvere alcuni di essi saranno sufficienti un paio d'ore di implementazione; altri invece sono dei veri e propri compiti in classe, che possono durare anche due o tre settimane di studio.

5.42 (*Analisi del testo*) La capacità dei computer di manipolare le stringhe ha avuto delle applicazioni interessanti nell'analisi dei testi dei grandi scrittori. C'è stato molto clamore intorno al dubbio, suscitato da questo tipo di analisi, se William Shakespeare sia mai esistito. Ci sono alcuni studiosi che considerano evidente la mano di Christopher Marlowe o di altri autori nelle opere tradizionalmente attribuite a Shakespeare. I ricercatori hanno utilizzato i computer per scoprire delle somiglianze tra diversi autori. In questo esercizio esamineremo tre metodi per analizzare un testo con il computer.

a) Scrivete un programma che legge diverse linee di testo dalla tastiera e visualizza una tabella con il numero di occorrenze di ogni lettera dell'alfabeto nel testo. Per esempio, la frase

   ```
   To be, or not to be: that is the question:
   ```

   contiene una "a", due "b", nessuna "c" e così via.

b) Scrivete un programma che legge diverse linee di testo dalla tastiera e visualizza una tabella con il numero di parole di una sola lettera, il numero di parole di due lettere, di tre lettere e così via. Per esempio la frase

   ```
   Whether 'tis nobler in the mind to suffer
   ```

   contiene

Lunghezza della parola	Occorrenze
1	0
2	2
3	2
4	2 ('tis compresa)
5	0
6	2
7	1

c) Scrivete un programma che legge diverse linee di testo e visualizza una tabella con il numero di occorrenze di ogni parola diversa. La prima versione del programma dovrebbe mostrare le parole nella tabella nello stesso ordine in cui appaiono nel testo. Per esempio, le linee

   ```
   To be, or not to be: that is the question:
   Whether 'tis nobler in the mind to suffer
   ```

   Contengono la parola "to" tre volte, la parola "be" due volte, la parola "or" una volta, e così via. Se volete rendere il programma più interessante, fategli visualizzare la tabella in ordine alfabetico.

5.43 (*Word processing*) Una funzione importante dei sistemi di elaborazione dei testi è la giustificazione del testo, cioè l'allineamento delle parole ai margini sinistro e destro della pagina. Ciò crea un documento dall'aspetto professionale, molto più di quello che accade con un documento scritto a macchina. La giustificazione del testo si può ottenere sui computer inserendo degli appositi spazi tra le parole, in modo tale che l'ultima parola di ogni riga termini sul margine destro.
Scrivete un programma che legge diverse linee di testo e visualizza il testo immesso con la giustificazione ai margini. Supponiamo che il testo debba essere stampato su un foglio del formato A4, e che

dobbiate mantenere il testo a una distanza di un paio di centimetri dai margini della pagina. Se il computer visualizza 4 caratteri per centimetro, il programma deve prevedere un massimo di 65 caratteri per linea.

5.44 (*Visualizzazione delle date nei vari formati*) Le date possono essere scritte in diversi formati, specie nella corrispondenza d'affari. Due formati molto comuni sono:

```
21/07/55 e 21 Luglio 1955
```

Scrivete un programma che legge una data nel primo formato e la trasforma nel secondo formato.

5.45 (*Sicurezza degli assegni circolari*) I computer sono impiegati spesso nelle operazioni di verifica degli assegni circolari e nelle applicazioni bancarie. Circolano, oltre agli assegni, anche delle strane storie su assegni emessi settimanalmente (per errore) per somme di 1 milione di dollari. I sistemi di emissione automatica degli assegni possono emettere degli assegni con somme scorrette, sia per un errore umano che della macchina. I progettisti di questi sistemi cercano di integrare dei controlli all'interno di essi, in modo da evitare l'emissione di assegni sbagliati.
Un altro problema piuttosto serio è l'alterazione intenzionale della somma scritta su un assegno, ad opera di persone fraudolente. Per scongiurare queste alterazioni, molti sistemi di emissione automatica di assegni utilizzano una tecnica di protezione, come quella che segue.
Gli assegni destinati a essere emessi dai computer prevedono un numero fisso di spazi dove il computer può stampare la somma di denaro. Supponiamo che un assegno preveda otto spazi bianchi in cui un computer deve scrivere l'ammontare di un assegno emesso con frequenza settimanale. Se la somma è piuttosto ingente, verranno riempiti tutti gli otto spazi, per esempio:

```
1,230.60   (somma)
--------
12345678   (posizioni scrivibili)
```

D'altra parte, se la somma è minore di 1.000 dollari, saranno molti gli spazi che restano in bianco. Per esempio

```
  99.87
--------
12345678
```

contiene tre spazi in bianco. Se questi spazi rimangono in bianco, sarà un gioco da ragazzi aggiungere qualche cifra e alterare la somma dell'assegno per qualche male intenzionato. Per prevenire questa alterazione, molti sistemi di emissione automatica inseriscono degli asterischi negli spazi iniziali che rimangono vuoti:

```
***99.87
--------
12345678
```

Scrivete un programma che riceve in input una somma in dollari da stampare su un assegno, e che visualizza la somma nel formato in cui comparirà sull'assgno, cioè nel formato protetto da asterischi (dove necessari). Supponete che gli spazi di stampa disponibili siano nove.

5.46 (*Scrivere in parole la somma di un assegno circolare*) Continuando sulla falsariga della discussione precedente, non perdiamo occasione per sottolineare l'importanza dei sistemi di controllo nell'emissione di assegni circolari. Per legge si richiederebbe, inoltre, che la somma sia scritta sia in numeri che a parole. Infatti se è vero che è semplice alterare le cifre dell'assegno, è estremamente difficile alterare la dicitura a parole.
Non tutti i sistemi di emissione automatica scrivono la dicitura a parole, probabilmente perché la maggior parte dei linguaggi ad alto livello utilizzati nelle applicazioni commerciali non possono manipolare le stringhe in modo adeguato. Un'altra ragione sta nel fatto che un programma di questo genere ricadrebbe nell'ambito della cosiddetta "traduzione automatica".

Scrivete un programma in C++ che riceve in input l'ammontare di un assegno e visualizza l'equivalente in parole della somma. Per esempio la somma 112.830 verrebbe visualizzata come

CENTODODICIMILAOTTOCENTOTRENTA

5.47 (*Codice Morse*) Forse il più famoso di tutti i sistemi di codifica di tutti i tempi è il codice Morse, ideato da Samuel Morse nel 1832 per il telegrafo. Il codice Morse assegna una serie di punti e linee a ogni lettera dell'alfabeto, a ogni numero e ad altri caratteri speciali, come il punto, la virgola, il due punti e il punto e virgola. Nei sistemi di comunicazione che si basavano sui suoni, il punto rappresentava un suono breve e la linea un suono lungo. Il sistema a punti e linee è stato poi tradotto anche nei sistemi di comunicazioni basati sulle luci e sui segnali.
La separazione tra le parole viene indicata da uno spazio, o semplicemente dall'assenza di punti e linee. Nel sistema dei suoni uno spazio è un periodo di silenzio di una certa durata. La versione internazionale del codice Morse è in Figura 5.42
Scrivete un programma che legge una frase e la codifica in codice Morse. Scrivete anche il programma opposto, che legge una frase in Morse e la traduce in linguaggio umano. Nella codifica Morse separate ogni lettera della stessa parola con uno spazio e due parole successive con tre spazi.

Carattere	Codice	Carattere	Codice
A	·−	T	−
B	−···	U	··−
C	−·−·	V	···−
D	−··	W	·−−
E	·	X	−··−
F	··−·	Y	−·−−
G	−−	Z	−−··
H	····		
I	··	*Cifre*	
J	·−−	1	·−−−
K	−·−	2	··−−
L	·−··	3	···−−
M	−−	4	····−
N	−·	5	·····
O	−−−	6	−····
P	·−−·	7	−−···
Q	−−·−	8	−−−··
R	·−·	9	−−−−·
S	···	0	−−−−

Figura 5.42 Le lettere dell'alfabeto espresse nel codice Morse internazionale.

5.48 (*Programma di conversione delle unità di misura*) Scrivete un programma che effettui le conversioni tra le varie unità di misura adottate comunemente. Il programma dovrebbe consentire all'utente di immettere il nome dell'unità di misura come una stringa (per es. centimetri, [illegible] sistema metrico decimale e pollici, quarti, galloni per quello anglosassone) e dovrebbe essere in grado di rispondere a semplici domande come

```
"Quanti pollici ci sono in 2 metri?"
"Quanti litri ci sono in 10 quarti?"
```

Il vostro programma dovrebbe anche saper reagire a conversioni non valide. Per esempio, la domanda

```
"Quanti piedi ci sono in 5 chilogrammi?"
```

non ha senso, perché i piedi misurano la lunghezza e i chilogrammi il peso.

Un progetto di manipolazione di stringhe complesso

5.49 (*Generatore di cruciverba*) La maggior parte di persone li risolve, ma sono in pochi coloro che si accingono a crearne: stiamo parlando dei cruciverba. Nel nostro caso si tratta di un progetto di manipolazione di stringhe che richiede un certo impegno, perché è piuttosto complesso. Dobbiamo risolvere molti problemi prima di poter vedere all'opera un generatore di cruciverba funzionante. Per esempio: come rappresentare internamente la griglia di un cruciverba? Meglio utilizzare una serie di stringhe o un array bidimensionale? Inoltre il programma ha bisogno di accedere a una sorgente di parole, cioè a un dizionario. In che formato conviene memorizzare questo dizionario, in modo che le manipolazioni del programma risultino semplificate? I più ambiziosi tra voi vorranno generare anche le definizioni Orizzontali e Verticali. In effetti la difficoltà del problema è tutta nella manipolazione delle stringhe: la programmazione dello schema non è particolarmente problematica.

CAPITOLO 6

Le classi e l'astrazione dei dati

Obiettivi

- Comprendere il concetto di astrazione dei dati e i tipi di dato astratti (ADT)
- Imparare a gestire i costrutti per la definizione di tipi di dato astratti in C++: le classi
- Comprendere come creare, utilizzare e distruggere gli oggetti di una classe
- Gestire il controllo dell'accesso ai dati e alle funzioni membro di un oggetto
- Comprendere l'importanza della programmazione orientata agli oggetti

6.1 Introduzione

A partire da questo capitolo introduciamo finalmente la programmazione orientata agli oggetti in C++. Il motivo per cui l'abbiamo fin qui rimandata sta principalmente nel fatto che gli oggetti che costruiremo sono composti in parte da segmenti strutturati di programmi, per cui dovevamo necessariamente gettare prima le basi della programmazione strutturata.

Nelle sezioni "Pensare in termini di oggetti" dei capitoli precedenti abbiamo introdotto i concetti fondamentali e la terminologia di base della programmazione a oggetti. Abbiamo poi parlato delle tecniche di *progettazione orientata agli oggetti* (*OOD*): ricorderete che nell'analisi della definizione del simulatore software di un ascensore abbiamo determinato quali erano le classi necessarie a implementare il sistema, assieme ad attributi e comportamenti, e infine abbiamo specificato che tipo di interazione doveva esistere tra i diversi oggetti perché il sistema nel suo complesso potesse funzionare correttamente.

Prima di addentrarci negli argomenti di questo capitolo, vogliamo ricordarvi brevemente i concetti più importanti e un po' di terminologia. La programmazione orientata agli oggetti (OOP) *incapsula* i dati (attributi) e le funzioni (comportamenti) in pacchetti detti *classi*: dati e funzioni sono strettamente correlati tra di loro. Una classe assomiglia a un progetto edilizio; se è in possesso di un progetto, un costruttore può dare inizio ai lavori. Si può osservare che con un solo progetto il costruttore può costruire quanti edifici desidera e, allo stesso modo, una classe può essere riutilizzata per creare più oggetti dello stesso tipo. Una proprietà delle classi è l'*occultamento delle informazioni*: infatti, se è vero che gli oggetti di classi diverse sono in grado di comunicare tra di loro tramite *interfacce* ben definite, di norma gli oggetti di una classe non conoscono i dettagli di implementazione degli oggetti delle altre classi. Questi dettagli rimangono nascosti, e sono custoditi all'inter-

no delle rispettive classi. Per fare un paragone con il mondo reale, è sicuramente possibile guidare un'automobile senza conoscere come funzionano il motore, la trasmissione e il sistema di alimentazione del carburante. Vedremo in seguito i motivi per i quali l'occultamento delle informazioni costituisce un argomento fondamentale dell'ingegneria del software.

In C e in altri linguaggi di programmazione procedurali la programmazione tende a essere *orientata all'azione* piuttosto che agli oggetti. In C l'unità fondamentale di programmazione è la funzione; in C++ questa unità diventa la classe, dalla quale si *istanziano* (creano) successivamente gli oggetti.

Chi programma in C deve concentrare la sua attenzione sulla scrittura delle funzioni. La procedura è in realtà abbastanza semplice: deve combinare in una funzione gruppi di operazioni simili, che effettuano un dato compito, e deve poi comporre il programma combinando le diverse funzioni. Naturalmente i dati sono importanti anche in C, ma sono visti principalmente come supporto alle operazioni delle funzioni. Chi programma in C andrà alla ricerca dei *verbi* presenti nella specifica di un sistema, per determinare l'insieme di funzioni che servono per implementare tale sistema.

Chi programma in C++, invece, deve creare i propri tipi di dato, detti *classi*. Le classi sono anche chiamate *tipi definiti dall'utente*. In ogni classe sono presenti dei dati e delle funzioni deputate a manipolare tali dati. I dati di una classe si chiamano *dati membro*. Le funzioni si chiamano *funzioni membro*, o *metodi*. L'istanza di un tipo predefinito come `int` si chiama *variabile*, mentre l'istanza di un tipo definito dall'utente, ovvero di una classe, si chiama *oggetto*. Ad ogni modo tenete presente che i programmatori utilizzano di norma i termini variabile e oggetto in modo equivalente. Dato che l'attenzione di chi programma in C++ si rivolge alle classi più che alle funzioni, hanno maggiore importanza i *nomi* anziché i verbi presenti nella specifica di un sistema, perché essi aiutano a determinare l'insieme di classi da progettare per implementare tale sistema.

Le classi del C++ sono la naturale evoluzione delle `struct` (strutture) del C. Prima di vedere come sono fatte le classi, quindi, parleremo delle strutture, e progetteremo tipi definiti dall'utente che si basano sulle strutture. Il concetto di struttura ha molti punti deboli, che cercheremo di illustrarvi man mano che li incontreremo; il concetto di classe cerca di superare proprio queste limitazioni.

6.2 Come si definisce una struttura

Le strutture sono tipi di dato aggregati che contengono elementi di altri tipi. La seguente è una tipica definizione di struttura:

```
struct Time {
   int hour;      // 0-23
   int minute;    // 0-59
   int second;    // 0-59
};
```

La parola riservata `struct` introduce la definizione della struttura. L'identificatore `Time` è l'*etichetta* della struttura. Essa dà il nome alla struttura e servirà, in seguito, a dichiarare variabili del tipo della struttura. In questo esempio il nome del nuovo tipo di dato è `Time`. I nomi dichiarati tra le parentesi graffe sono i *membri* della struttura. I membri di una stessa struttura devono avere nomi univoci, ma due strutture diverse possono contenere membri che hanno lo stesso nome, perché ciò non causa un conflitto. La definizione di una struttura deve sempre terminare con un punto e virgola. Come vedremo, queste convenzioni valgono anche per le classi, perché strutture e classi sono piuttosto simili in C++.

La struttura `Time` contiene tre membri di tipo `int`, e cioè `hour`, `minute` e `second`. I membri di una struttura possono essere di qualunque tipo, e in realtà si utilizzano spesso strutture che contengono membri di molti tipi diversi. L'unica restrizione è che una struttura non può contenere un'istanza di se stessa: quello che intendiamo è che un membro di `Time` non può essere una variabile di tipo `Time`. Tuttavia è possibile includere un puntatore a un'altra struttura `Time`. Una struttura che contiene un puntatore allo stesso tipo di struttura si chiama *struttura autoreferente*. Come vedremo nel Capitolo 4 del volume Tecniche Avanzate, questo tipo di struttura è molto utile per formare strutture dati concatenate, come liste, code, pile e alberi.

La definizione di una struttura non riserva dello spazio in memoria: il suo scopo è creare un nuovo tipo di dato, che servirà per dichiarare un nuovo tipo di variabili. La variabili di struttura si dichiarano come tutte le altre. La dichiarazione

```
Time timeObject, timeArray[ 10 ], *timePtr
   &timeRef = timeObject;
```

dichiara `timeObject` come variabile di tipo `Time`, `timeArray` come array di 10 elementi di tipo `Time`, `timePtr` come puntatore a un oggetto `Time` e `timeRef` come riferimento a un oggetto `Time` inizializzata con `timeObject`.

6.3 Come si accede ai membri di una struttura

Per accedere ai membri di una struttura (o di una classe) si utilizzano gli *operatori di accesso ai membri*, che sono l'*operatore punto* (`.`) e l'*operatore freccia* (`->`). L'operatore punto accede a un membro di una struttura o di una classe tramite il nome di variabile dell'oggetto o un riferimento all'oggetto. Per esempio, per visualizzare il valore del membro `hour` della struttura `timeObject` si può utilizzare l'istruzione

```
cout << timeObject.hour;
```

Per visualizzare il membro `hour` della struttura riferita da `timeRef` si può utilizzare l'istruzione

```
cout << timeRef.hour;
```

L'operatore freccia, che è composto da un segno meno (`-`) e un segno di maggiore (`>`) scritti di seguito senza spazi di separazione, accede a un membro di una struttura o di una classe tramite un puntatore all'oggetto. Supponiamo che il puntatore `timePtr` punti a un oggetto `Time` e che contenga l'indirizzo di `timeObject`. Per visualizzare il membro `hour` di `timeObject` tramite il puntatore `timePtr` basta utilizzare l'istruzione

```
timePtr = &timeObject;
cout << timePtr->hour;
```

L'espressione `timePtr->hour` equivale a `(*timePtr).hour`, in quanto dereferenzia il puntatore per accedere al membro `hour`. Le parentesi sono necessarie, perché l'operatore punto ha precedenza più alta dell'operatore di risoluzione del riferimento (`*`). L'operatore freccia e l'operatore punto hanno precedenza maggiore di tutti gli altri operatori, dopo le parentesi tonde e quadre, e associano da sinistra a destra.

Errore tipico 6.1

L'espressione `(*timePtr).hour` *riferisce il membro* `hour` *della* `struct` *a cui punta* `timePtr`. *Se dimenticate di scrivere le parentesi e scrivete* `*timePtr.hour`, *commettete un errore di sintassi, perché la precedenza di* `.` *è più alta di quella di* `*`, *per cui il compilatore interpreterà l'espressione come* `*(timePtr.hour)`. *Questa istruzione contiene un errore di sintassi, perché l'operatore* `.` *non si applica ai puntatori. La scrittura corretta prevede l'uso dell'operatore freccia in questo modo:* `timePtr->hour`.

6.4 L'implementazione del tipo di dato Time come struttura

Il programma in Figura 6.1 crea il tipo definito dall'utente `Time` che ha tre membri interi: `hour`, `minute` e `second`. Il programma definisce una sola struttura `Time` di nome `dinnerTime`, e utilizza l'operatore punto per inizializzare i membri della struttura con i seguenti valori: `18` per `hour`, `30` per `minute` e `0` per `second`. Il programma visualizza un orario nel cosiddetto *formato universale* (che negli Stati Uniti viene anche chiamato formato militare) e nel *formato standard* (che rappresenta gli orari usando 12 ore e i suffissi AM e PM per le ore antimeridiane e pomeridiane rispettivamente). Come notate, le funzioni `print-` ricevono riferimenti a strutture `Time` costanti. In questo modo le strutture `Time` sono passate alla funzioni `print-` per riferimento, eliminando il tempo necessario per effettuare prima la copia necessaria alla chiamata per valore. L'utilizzo di `const` vieta la modifica della struttura `Time` all'interno delle funzioni. Nel Capitolo 7 torneremo a parlare degli oggetti `const` e delle funzioni membro `const`.

```
1  // Fig. 6.1: fig06_01.cpp
2  // Crea una struttura, ne imposta i membri e la visualizza.
3  #include <iostream.h>
4
5  struct Time {     // definizione della struttura
6     int hour;      // 0-23
7     int minute;    // 0-59
8     int second;    // 0-59
9  };
10
11 void printMilitary( const Time & );  // prototipo
12 void printStandard( const Time & );  // prototipo
13
14 int main()
15 {
16    Time dinnerTime;    // una variabile del nuovo tipo Time
17
18    // imposta i membri con valori validi
19    dinnerTime.hour = 18;
20    dinnerTime.minute = 30;
21    dinnerTime.second = 0;
```

Figura 6.1 Un programma che utilizza la struttura Time (continua)


```
22
23    cout << "Dinner will be held at ";
24    printMilitary( dinnerTime );
25    cout << " military time,\nwhich is ";
26    printStandard( dinnerTime );
27    cout << " standard time.\n";
28
29    // imposta i membri con valori non validi
30    dinnerTime.hour = 29;
31    dinnerTime.minute = 73;
32
33    cout << "\nTime with invalid values: ";
34    printMilitary( dinnerTime );
35    cout << endl;
36    return 0;
37 }
38
39 // Visualizza l'orario in formato militare
40 void printMilitary( const Time &t )
41 {
42    cout << ( t.hour < 10 ? "0" : "" ) << t.hour << ":"
43         << ( t.minute < 10 ? "0" : "" ) << t.minute;
44 }
45
46 // Visualizza l'orario in formato standard
47 void printStandard( const Time &t )
48 {
49    cout << ( ( t.hour == 0 || t.hour == 12 ) ?
50              12 : t.hour % 12 )
51         << ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
52         << ":" << ( t.second < 10 ? "0" : "" ) << t.second
53         << ( t.hour < 12 ? " AM" : " PM" );
54 }
```

```
Dinner will be held at 18:30:00 military time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73
```

Figura 6.1 Un programma che utilizza la struttura Time.

Obiettivo efficienza 6.1

Per default le strutture sono passate alle funzioni per valore. Per eliminare il tempo necessario alla copia tipico di questo tipo di chiamata, utilizzate la chiamata per riferimento.

Ingegneria del software 6.1

Se volete evitare una chiamata per valore, ma volete anche proteggere i dati originari della funzione chiamante, passate gli argomenti di grosse dimensioni come riferimenti `const`.

La creazione di nuovi tipi di dato in questo modo comporta anche degli inconvenienti. Dato che non è richiesta un'inizializzazione esplicita, è possibile dover affrontare i tipici problemi dei dati non inizializzati. Inoltre, anche se i dati sono effettivamente inizializzati, i valori possono non essere corretti. Infatti ai membri di una struttura il programma può assegnare anche valori non validi (come accade in Figura 6.1) perché può accedere direttamente ai dati. Nelle righe 30 e 31, il programma ha assegnato dei valori senza significato ai membri `hour` e `minute` dell'oggetto `dinnerTime`. Un altro inconveniente sta nel fatto che, se si decide di modificare l'implementazione della `struct`, rappresentando per esempio l'orario come numero di secondi passati dall'ultima mezzanotte, si devono apportare delle modifiche anche in tutti i programmi che utilizzano tale `struct`. Il motivo è ancora una volta il fatto che il programmatore manipola direttamente il tipo di dato. Non c'è alcuna "interfaccia" che assicuri che il programmatore utilizzi correttamente il dato di tale tipo, e che questo dato abbia sempre un significato.

Ingegneria del software 6.2

Sforzatevi sempre di scrivere programmi comprensibili e semplici da mantenere. Le modifiche sono la regola, piuttosto che l'eccezione. Scrivete il vostro codice tenendo presente che probabilmente lo modificherete. Come vedremo, le classi rendono i programmi più semplici da modificare.

Le strutture comportano anche un'altra serie di problemi. In C le strutture non possono essere visualizzate come unità singola, ma si può soltanto visualizzare e formattare ciascuno dei membri che le compongono. Una soluzione sta nello scrivere una funzione apposita che visualizzi i membri di una struttura nell'ordine e nel formato appropriato. Nel Capitolo 8 mostreremo come effettuare l'overloading dell'operatore `<<` per visualizzare facilmente gli oggetti di una struttura o di una classe. In C, inoltre, le strutture non possono essere confrontate direttamente, ma soltanto membro a membro. L'overloading degli operatori relazionali e di uguaglianza risolve anche questo problema, consentendo di confrontare direttamente gli oggetti di strutture e classi.

La prossima sezione implementa `Time` come classe anziché come struttura, e mostra alcuni dei vantaggi dei tipi di dato astratti. In C++ classi e strutture possono essere utilizzati in modo quasi equivalente. La differenza tra le due forme sta nell'accessibilità di default dei membri; torneremo su questo punto tra breve.

6.5 L'implementazione del tipo di dato Time come classe

Le classi consentono di modellare oggetti che hanno attributi (dati membro) e comportamenti (funzioni membro). I tipi di dato che contengono dati e funzioni membro si definiscono con la parola riservata `class`.

In altri linguaggi orientati agli oggetti le funzioni membro si chiamano anche *metodi*, e sono invocate quando un oggetto riceve un messaggio. Un messaggio corrisponde alla chiamata di una funzione membro, effettuata da un altro oggetto o da una funzione.

Quando si definisce una classe, il suo nome può essere utilizzato per dichiarare oggetti di tale classe. In Figura 6.2 mostriamo una semplice definizione della classe `Time`.

La definizione di `Time` inizia con la parola chiave `class`. Il corpo della definizione è racchiuso tra due parentesi graffe e l'intera definizione termina con un punto e virgola. Sia la classe `Time` che la struttura `Time` contengono i tre interi `hour`, `minute` e `second`.

```
1   class Time {
2   public:
3      Time();
4      void setTime( int, int, int );
5      void printMilitary();
6      void printStandard();
7   private:
8      int hour;     // 0 - 23
9      int minute;   // 0 - 59
10     int second;   // 0   59
11  };
```

Figura 6.2 La definizione della classe Time.

Errore tipico 6.2

Se dimenticate il punto e virgola che termina la definizione di una classe o di una struttura, commettete un errore di sintassi.

Tutto il resto della definizione vi risulterà sicuramente nuovo. Le etichette `public:` e `private:` si chiamano *specificatori di accesso ai membri*. I dati e le funzioni membro che sono elencati dopo lo specificatore `public` (e prima dello specificatore successivo) sono accessibili in tutti i punti del programma in cui si può accedere a un oggetto della classe `Time`. I dati e le funzioni membro che sono elencati dopo lo specificatore `private` sono accessibili invece soltanto alle funzioni membro della classe. Gli specificatori di accesso sono sempre seguiti da un carattere di due punti e possono comparire più volte e in qualsiasi ordine nella definizione di una classe. Nel seguito faremo riferimento agli specificatori `public` e `private` senza utilizzare il segno di due punti e utilizzeremo i termini italiani pubblico e privato come loro sinonimi. Nel Capitolo 9 studieremo un terzo specificatore di accesso, `protected` che è legato ad un meccanismo di programmazione per estensione chiamato ereditarietà.

Buona abitudine 6.1

Scrivete ciascuno specificatore di accesso una sola volta nella definizione di una classe, per migliorare la leggibilità del vostro codice. I membri `public` dovrebbero essere posti per primi in modo da localizzarli più facilmente.

La definizione di `Time` contiene i prototipi di queste quattro funzioni membro pubbliche: `Time`, `setTime`, `printMilitary` e `printStandard`. I dati e le funzioni membro `public` sono anche detti *interfaccia* della classe. I *client*, cioè le porzioni del programma che utilizzano la classe, utilizzeranno queste funzioni per manipolare i dati della classe.

Se osservate, c'è una funzione membro che ha lo stesso nome della classe: questa funzione prende il nome di *costruttore della classe*. Un costruttore è una funzione membro speciale che inizializza i dati membro di un oggetto della classe. Quando viene creato un oggetto di una classe, il costruttore di tale classe viene invocato automaticamente. Come

vedremo, è perfettamente normale che una classe abbia più di un costruttore e ciò è possibile grazie al meccanismo di overloading delle funzioni. Osservate che il costruttore non restituisce alcun tipo di dato.

Errore tipico 6.3

Un costruttore non restituisce alcun tipo di dato: se ne specificate uno commettete un errore di sintassi.

I tre membri interi di `Time` compaiono dopo lo specificatore di accesso `private`. Ciò significa che essi saranno accessibili soltanto alle funzioni membro di `Time` e, come vedremo, alle funzioni `friend` (amiche) della classe. Quindi possono accedere a questi dati solo le quattro funzioni della classe. I dati membro sono normalmente elencati nella porzione `private` della classe, mentre le funzioni membro nella porzione `public`. Tuttavia non c'è alcun divieto di dichiarare funzioni `private` e dati `public`, e lo vedremo tra breve, ma l'obiezione comune è che non si tratti di una buona abitudine di programmazione.

Dopo aver definito la classe, è possibile utilizzarla come tipo di dato nelle dichiarazioni, come segue:

```
Time sunset,                    // oggetto di tipo Time
     arrayOfTimes[ 5 ],         // array di oggetti Time
     *pointerToTime,            // puntatore a un oggetto Time
     &dinnerTime = sunset;      // riferimento a un oggetto Time
```

Il nome della classe è adesso un nuovo specificatore di tipo. Possiamo creare molti oggetti della stessa classe, così come possiamo, ad esempio, creare molte variabili di tipo `int`. Inoltre la possibilità di creare nuove classi, a seconda delle esigenze del problema, rende il C++ un linguaggio versatile ed estensibile.

Il programma in Figura 6.3 fa uso della classe `Time`. Questo programma istanzia un solo oggetto della classe `Time` di nome `t`. Quando viene creato l'oggetto, viene anche invocato automaticamente il costruttore di `Time` che inizializza esplicitamente tutti i dati membro a `0`. Il programma visualizza poi l'orario in formato militare e in formato standard, per segnalare che i dati sono stati inizializzati correttamente. Successivamente la funzione `setTime` imposta l'orario, e questo viene visualizzato ancora una volta nei due formati. Infine `setTime` tenta di assegnare ai dati membro dei valori non validi, e l'orario viene visualizzato un'ultima volta.

```
1   // Fig. 6.3: fig06_03.cpp
2   // La classe Time.
3   #include <iostream.h>
4
5   // Definizione di Time come tipo di dato astratto (ADT)
6   class Time {
7   public:
8      Time();                          // costruttore
9      void setTime( int, int, int);    // imposta hour,minute,second
10     void printMilitary();            // visualizza l'orario in
11                                      // formato militare
```

Figura 6.3 Implementazione di un tipo di dato astratto: la classe Time (continua)


```
12     void printStandard();    // visualizza l'orario in
13                              // formato standard
14  private:
15     int hour;      // 0   23
16     int minute;    // 0   59
17     int second;    // 0   59
18  };
19
20  // Il costruttore di Time inizializza ogni dato
21  // membro a zero.
22  // Assicura che tutti gli oggetti Time all'inizio siano
23  // in uno stato coerente.
24  Time::Time() { hour = minute = second = 0; }
25
26  // Imposta un nuovo valore di Time utilizzando il formato
27  // militare. Controlla la validita
28  // dei valori dei dati. Imposta i valori non validi a zero.
29  void Time::setTime( int h, int m, int s )
30  {
31     hour = ( h >= 0 && h < 24 ) ? h : 0;
32     minute = ( m >= 0 && m < 60 ) ? m : 0;
33     second = ( s >= 0 && s < 60 ) ? s : 0;
34  }
35
36  // Print Time in military format
37  void Time::printMilitary()
38  {
39     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
40          << ( minute < 10 ? "0" : "" ) << minute;
41  }
42
43  // Visualizza l'orario in formato standard
44  void Time::printStandard()
45  {
46     cout << ( ( hour == 0 || hour == 12 ) ? 12   hour % 12 )
47          << ":" << ( minute < 10 ? "0" : "" ) << minute
48          << ":" << ( second < 10 ? "0"   "" ) << second
49          << ( hour < 12 ? " AM" : " PM" );
50  }
51
52  // Programma di esempio per la classe Time
53  int main()
54  {
55     Time t;  // istanzia l'oggetto t della classe Time
56
57     cout << "The initial military time is ";
58     t.printMilitary();
```

Figura 6.3 Implementazione di un tipo di dato astratto: la classe Time (continua)

```
59      cout << "\nThe initial standard time is ";
60      t.printStandard();
61
62      t.setTime( 13, 27, 6 );
63      cout << "\n\nMilitary time after setTime is ";
64      t.printMilitary();
65      cout << "\nStandard time after setTime is ";
66      t.printStandard();
67
68      t.setTime( 99, 99, 99 );      // tentativo di immettere
69                                    // valori non validi
70      cout << "\n\nAfter attempting invalid settings:"
71           << "\nMilitary time: ";
72      t.printMilitary();
73      cout << "\nStandard time: ";
74      t.printStandard();
75      cout << endl;
76      return 0;
77  }
```

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

Figura 6.3 Implementazione di un tipo di dato astratto: la classe Time .

Vale la pena di notare nuovamente che i dati membro `hour`, `minute` e `second` sono membri `private` perché sono preceduti dallo specificatore di accesso corrispondente. Come abbiamo già detto, i membri `private` di una classe non sono accessibili normalmente dall'esterno di essa, fatta eccezione per le funzioni `friend` di cui parleremo nel Capitolo 7.

La filosofia che sta dietro a ciò è che ai client di una classe non deve interessare il modo in cui sono rappresentati i dati in tale classe.

Per esempio, sarebbe perfettamente lecito se decidessimo di rappresentare internamente l'orario della classe `Time` come numero di secondi passati dall'ultima mezzanotte. I client potranno continuare a utilizzare le stesse funzioni membro, ottenendo gli stessi risultati di prima, senza conoscere questo nuovo dettaglio. In questo senso l'implementazione di una classe è *nascosta* ai client.

Ciò rende i programmi più semplici da modificare e semplifica, in un certo senso, la percezione che hanno i client delle classi che utilizzano.

Ingegneria del software 6.3

I client di una classe utilizzano tale classe senza conoscere i dettagli dell'implementazione interna della classe. Se si decide di modificare l'implementazione interna, ad esempio per ottimizzarla, ma l'interfaccia della classe resta inalterata, il codice sorgente dei client della classe non ha bisogno di subire alcuna modifica, anche se probabilmente andrà ricompilato. Ciò rende i sistemi più semplici da aggiornare e modificare.

In questo programma il costruttore di `Time` inizializza semplicemente i dati membro a `0`, ovvero a mezzanotte nella rappresentazione militare. In questo modo possiamo star certi che l'oggetto sarà in uno stato coerente quando viene creato. Non è possibile memorizzare valori non validi nei dati membro di un oggetto `Time`, perché alla sua creazione viene chiamato automaticamente il costruttore, mentre tutti i successivi tentativi dei client di modificare l'orario devono passare il vaglio della funzione `setTime`.

Ingegneria del software 6.4

Le funzioni membro sono generalmente più concise delle funzioni dei programmi non orientati agli oggetti, perché i dati membro si considerano già convalidati da un costruttore e/o da ogni altra funzione membro che opera su di essi. Inoltre i dati sono già presenti nell'oggetto, per cui le funzioni membro raramente hanno bisogno di molti argomenti, al contrario delle funzioni non orientate agli oggetti. Di conseguenza risultano anche più concise le chiamate, le definizioni e i prototipi delle funzioni.

I dati membri di una classe non possono essere inizializzati durante la loro dichiarazione nel corpo della classe. A questo scopo bisogna utilizzare il costruttore della classe o altre funzioni apposite.

Errore tipico 6.4

Se cercate di inizializzare un dato membro della classe esplicitamente durante la definizione della classe commettete un errore di sintassi.

Una funzione che ha lo stesso nome della classe ed è preceduta dal carattere tilde (~) si chiama *distruttore* della classe. In mancanza di un distruttore esplicito, il sistema ne crea automaticamente uno di default. Il distruttore effettua le operazioni finali prima che la memoria occupata da un oggetto sia restituita al sistema. I distruttori non accettano argomenti, e quindi non possono neanche subire overloading. Parleremo più in dettaglio dei costruttori e dei distruttori nel seguito di questo capitolo e nel Capitolo 7.

Le funzioni definite per essere usate dall'esterno sono precedute dall'etichetta public. Queste funzioni implementano i comportamenti (o *servizi*) che la classe fornisce ai propri client, e costituiscono la cosiddetta *interfaccia pubblica* della classe.

Ingegneria del software 6.5

I client di una classe possono accedere all'interfaccia della classe ma non ai dettagli della sua implementazione.

La definizione di una classe contiene le dichiarazioni di dati membro e funzioni membro. In una dichiarazione si trovano generalmente soltanto i prototipi delle funzioni membro: infatti anche se queste possono essere definite all'interno della classe, si considera buona abitudine definirle all'esterno della definizione di classe.

Ingegneria del software 6.6

Se dichiarate le funzioni membro nella definizione di una classe (con i prototipi) e le definite all'esterno della classe, separate l'interfaccia della classe dalla sua implementazione. Si tratta di una buona abitudine di programmazione: i client della classe non saranno in grado di conoscere l'implementazione delle funzioni membro.

Come vedete, in Figura 6.3 abbiamo utilizzato l'*operatore binario di risoluzione dello scope* (`::`) in ciascun membro che segue la definizione di classe. Quando si dichiara una classe, bisogna definire poi le sue funzioni membro. Ognuna di esse può essere definita direttamente nel corpo della classe, anziché includerne soltanto il prototipo, oppure può essere definita al di fuori di esso. Nell'ultimo caso la definizione deve contenere anche il nome della classe e l'operatore binario di risoluzione dello scope (`::`). Potrebbero esserci infatti classi diverse con membri dello stesso nome, per cui l'operatore di risoluzione dello scope consente di riferirsi in modo univoco il nome della funzione a una classe particolare.

Errore tipico 6.5

Se decidete di definire una funzione membro al di fuori della classe, ricordate di includere il nome della classe e l'operatore binario di risoluzione dello scope, altrimenti commettete un errore di sintassi.

Naturalmente anche se definite una funzione membro al di fuori della sua classe, rimane sempre all'interno dello scope della classe, cioè il suo nome è noto soltanto agli altri membri della classe, a meno che non lo si utilizza tramite un oggetto, un riferimento a un oggetto o un puntatore a un oggetto della classe. Torneremo sullo scope a livello di classe tra breve.

Se si definisce una funzione membro in una definizione di classe, essa diventa automaticamente una funzione `inline`. Invece le funzioni membro definite al di fuori della definizione di classe possono essere rese `inline` utilizzando la parola chiave `inline`. Ad ogni modo ricordate che il compilatore decide volta per volta se è il caso di rendere `inline` una funzione, anche in presenza della parola chiave `inline`.

Obiettivo efficienza 6.2

Se definite una piccola funzione membro all'interno della definizione di una classe la renderete automaticamente `inline`, se il compilatore decide che è il caso. L'efficienza sarà superiore, ma ciò andrà a discapito di una buona metodologia di progettazione, perché i dettagli di implementazione della funzione saranno visibili ai client.

Ingegneria del software 6.7

Definite nell'intestazione della classe soltanto le funzioni membro più elementari.

È interessante che `printMilitary` e `printStandard` non prendano argomenti: ciò accade perché sanno implicitamente che devono visualizzare i dati membro dell'oggetto particolare per cui sono invocate. In questo modo si abbreviano e si semplificano notevolmente le chiamate di funzione, rispetto a quanto succede nella programmazione procedurale in vecchio stile.

Collaudo e messa a punto 6.1

Diminuendo il numero di argomenti da passare alle funzioni membro, diminuiscono anche le probabilità di passare argomenti scorretti, del tipo sbagliato e/o in numero sbagliato.

Ingegneria del software 6.8

Uno dei pregi della programmazione orientata agli oggetti è che le chiamate di funzione risultano semplificate, perché diminuisce il numero di argomenti necessari. Questo pregio deriva dall'incapsulamento dei dati e delle funzioni in un solo oggetto, per cui le funzioni sanno accedere ai dati di cui hanno bisogno.

Le classi semplificano la programmazione, perché il client deve solo occuparsi delle operazioni incapsulate in un oggetto. Queste operazioni sono progettate di norma pensando già all'uso che ne faranno i client; essi non devono preoccuparsi dei dettagli di implementazione di una classe, pur pretendendo che questa sia ovviamente corretta ed efficiente. Le interfacce possono certamente cambiare, ma ciò accade molto più di rado rispetto all'implementazione interna. Se occorre modificare l'implementazione, si deve agire conseguentemente anche sulle parti di codice che dipendono da essa. Mantenendo nascosta l'implementazione, non è possibile creare programmi che dipendano dai dettagli interni di una classe.

Ingegneria del software 6.9

Uno dei temi ricorrenti di questo testo è il "riutilizzo del software". Discuteremo attentamente di numerose tecniche per migliorare le classi e renderle riutilizzabili in diversi programmi. Per quanto ci riguarda, teniamo molto a insegnarvi come creare classi che vadano a costituire un "patrimonio software" di valore.

Non sempre bisogna creare una classe ex novo: è possibile derivare una nuova classe da altre classi, fornendole ulteriori attributi e comportamenti. Oppure è possibile progettare una nuova classe che contenga come membri oggetti di altre classi. In questo modo si migliora la produttività del processo di sviluppo. Quando si deriva una nuova classe da altre già esistenti si parla di *ereditarietà*, concetto che sarà presentato nel Capitolo 9. Quando in una classe si includono oggetti di altre classi si parla di *composizione*, e ne discuteremo nel prossimo capitolo.

Chi si avvicina per la prima volta alla programmazione orientata agli oggetti spesso esprime la preoccupazione che gli oggetti possano diventare presto abbastanza voluminosi, dal momento che contengono dati e funzioni. Da un punto di vista logico ciò è vero, perché si può considerare un oggetto come un aggregato di dati e funzioni, ma non è la stessa cosa dal punto di vista fisico.

Obiettivo efficienza 6.3

Un oggetto contiene in realtà soltanto i dati, per cui è molto più piccolo di quanto non sarebbe se contenesse anche le funzioni. Se applicate l'operatore `sizeof` al nome di una classe o a un suo oggetto, avrete come risultato soltanto la somma delle dimensioni dei dati. Il compilatore infatti crea una sola copia delle funzioni membro, separatamente dal resto degli oggetti della classe: tutti gli oggetti della classe devono condividere quest'unica copia delle funzioni. Ovviamente ogni oggetto ha bisogno della propria copia dei dati, perché questi sono diversi da oggetto a oggetto. Il codice delle funzioni invece è identico per tutti gli oggetti (spesso viene detto anche codice rientrante *o* procedura pura*), per cui può essere messo in condivisione.*

6.6 La visibilità a livello di classe e l'accesso ai membri di una classe

I dati e le funzioni membro di una classe appartengono allo scope di tale classe. Le funzioni non membro sono definite con scope a livello di file.

All'interno dello scope di una classe i membri sono immediatamente accessibili a tutte le funzioni membro della classe, e possono essere riferiti semplicemente con il proprio nome. Fuori dello scope di quella classe, i membri sono riferiti tramite un *handle* di un oggetto, cioè il suo nome oppure un riferimento o un puntatore ad esso. Nel Capitolo 7 vedremo che il compilatore inserisce un handle implicito per ogni dato o funzione membro di un oggetto.

Le funzioni membro di una classe possono essere sovrascritte, ma soltanto da altre funzioni membro della stessa classe. Per effettuare l'overloading di una funzione basta scrivere nella definizione della classe il prototipo di ogni sua versione, scrivendo in seguito i rispettivi codici di tali versioni.

Le funzioni membro hanno scope di funzione in una classe: le variabili definite in una funzione membro sono note soltanto a quella funzione. Se una funzione membro definisce una variabile che ha lo stesso nome di una variabile nello scope di classe, quest'ultima risulta nascosta dalla prima all'interno dello scope di funzione. Tuttavia è ancora possibile accedere alla variabile con scope di classe, utilizzando il nome della classe con l'operatore binario di risoluzione dello scope (`::`). L'operatore unario di risoluzione dello scope serve invece ad accedere alle variabili globali (cfr. Capitolo 3).

Per accedere ai membri di una classe si utilizzano gli stessi operatori delle strutture. L'operatore punto (`.`) si deve combinare con il nome di un oggetto o con un riferimento a un oggetto, mentre l'operatore freccia (`->`) con un puntatore a un oggetto.

Il programma in Figura 6.4 opera su una semplice classe, `Count`, con il dato membro intero `x` di tipo `public`, e la funzione `print` di tipo `public`. Il programma illustra l'uso degli operatori punto e freccia. Sono istanziate all'inizio tre variabili di tipo `Count`, e cioè `counter`, `counterRef` (un riferimento a un oggetto `Count`) e `counterPtr` (un puntatore a un oggetto `Count`). La variabile `counterRef` viene definita come riferimento a `counter`, mentre la variabile `counterPtr` come puntatore a `counter`. È importante notare che abbiamo reso `public` il dato membro `x` per mostrare come si accede ai membri `public` tramite un handle, cioè tramite un nome, un riferimento o un puntatore. Tuttavia ribadiamo il concetto che i dati membro sono generalmente `private`, e così sarà nella maggior parte dei nostri programmi.

```
1   // Fig. 6.4: fig06_04.cpp
2   // Programma che illustra gli operatori di accesso ai
3   // membri . e ->
4   // ATTENZIONE. NEI PROSSIMI ESEMPI EVITEREMO I DATI PUBLIC!
5   #include <iostream.h>
6
7   // La semplice classe Count
```

Figura 6.4 Accesso a dati e funzioni membro di un oggetto tramite i tre tipi di handle: nome, riferimento e puntatore a un oggetto (continua)


```
8   class Count {
9   public:
10     int x;
11     void print() { cout << x << endl; }
12  };
13
14  int main()
15  {
16     Count counter,                // crea un oggetto counter
17           *counterPtr = &counter, // puntatore a counter
18           &counterRef = counter;  // riferimento a counter
19
20     cout << "Assign 7 to x and print using the object's name: ";
21     counter.x = 7;          // assegna 7 al dato membro x
22     counter.print();        // chiama la funzione membro print
23
24     cout << "Assign 8 to x and print using a reference: ";
25     counterRef.x = 8;       // assegna 8 al dato membro x
26     counterRef.print();     // chiama la funzione membro print
27
28     cout << "Assign 10 to x and print using a pointer: ";
29     counterPtr->x = 10;     // assegna 10 al dato membro x
30     counterPtr->print();    // chiama la funzione membro print
31     return 0;
32  }
```

```
Assign 7 to x and print using the object's name: 7
Assign 8 to x and print using a reference: 8
Assign 10 to x and print using a pointer: 10
```

Figura 6.4 Accesso a dati e funzioni membro di un oggetto tramite i tre tipi di handle: nome, riferimento e puntatore a un oggetto.

6.7 La separazione di interfaccia e implementazione

Mantenere distinte interfaccia e implementazione costituisce uno dei principi fondamentali dell'ingegneria del software. I programmi scritti in questo modo sono più semplici da modificare. Per quel che riguarda le classi, le modifiche all'implementazione della classe non hanno alcuna influenza sui client di quella classe, sempre che non si modifichi anche l'interfaccia: ciò significherebbe che le funzionalità della classe sono estese al di là di quanto prevedeva l'interfaccia originaria.

Ingegneria del software 6.10

Ponete la dichiarazione di una classe in un file di intestazione e includete questo file in ogni client che la utilizza. Tale file costituirà l'interfaccia pubblica della classe e il client avrà a disposizione i prototipi necessari per chiamare le funzioni membro della classe. Ponete poi le definizioni delle funzioni membro in un file sorgente. Tale file costituirà l'implementazione della classe.

Ingegneria del software 6.11

I client di una classe non hanno bisogno di accedere al codice sorgente della classe, ma al suo codice oggetto durante la fase di linking. Questo fatto incoraggia i produttori di software a vendere o a concedere su licenza le proprie librerie di classi. Essi forniscono soltanto i file di intestazione e i moduli oggetto senza rivelare le informazioni di implementazione, come succederebbe se fosse necessario fornire il codice oggetto. Le librerie di classi dei diversi produttori costituiscono un patrimonio per l'intera comunità di programmatori.

Nella realtà le cose non sono però mai perfette. I file di intestazione contengono qualche porzione di implementazione, anche soltanto qualche cenno. Le funzioni membro `inline`, per esempio, devono trovarsi necessariamente in un file di intestazione, perché durante la compilazione di un client il compilatore possa sostituire tutte le occorrenze della funzione con il codice della funzione.

I membri `private` sono presenti nella definizione di classe in un file di intestazione, per cui sono visibili ai client anche se questi non possono accedervi. Nel Capitolo 7 mostreremo come utilizzare una *classe proxy* per nascondere i dati `private` ai client della classe.

Ingegneria del software 6.12

Le informazioni importanti per l'interfaccia di una classe devono essere incluse nel file di intestazione. Le informazioni che servono soltanto per gli scopi interni della classe e che non sono utili ai client dovrebbero essere messe in un file sorgente da non rendere pubblico. Questo è un altro esempio del principio del minor privilegio.

Il programma in Figura 6.5 è il programma di Figura 6.3 suddiviso in più file. Quando si crea un programma, la definizione di ogni classe viene posta normalmente in un file di intestazione, mentre le definizioni delle funzioni membro sono poste in un file sorgente con stesso nome. I file di intestazione saranno poi inclusi (con `#include`) in ogni file in cui sarà utilizzata la classe, mentre il file sorgente sarà compilato e collegato con il file che contiene il programma principale. Consultate la documentazione del vostro compilatore per sapere come compilare ed effettuare il linking dei programmi composti da più file sorgente.

La Figura 6.5 mostra il file di intestazione `time1.h` in cui viene dichiarata la classe `Time`, il file `time1.cpp` in cui sono definite le funzioni membro della classe `Time` e il file `fig06_05.cpp` in cui è definita la funzione `main`. *L'output di questo programma è uguale* a quello in Figura 6.3.

```
1   // Fig. 6.5: time1.h
2   // Dichiarazione della classe Time.
3   // Le funzioni membro sono definite in time1.cpp
4
5   // evita di includere più volte il file di intestazione
6   #ifndef TIME1_H
7   #define TIME1_H
8
9   // Definizione di tipo di dato astratto
```

Figura 6.5 Separazione dell'interfaccia e dell'implementazione per la classe Time (continua)


```
10  class Time {
11  public:
12     Time();                          // costruttore
13     void setTime( int, int, int ); // imposta hour, minute,second
14     void printMilitary();  // visualizza l'ora in formato militare
15     void printStandard();  // visualizza l'ora in formato standard
16  private:
17     int hour;      // 0   23
18     int minute;    // 0   59
19     int second;    // 0   59
20  };
21
22  #endif

23  // Fig. 6.5: time1.cpp
24  // Definizioni delle funzioni membro della classe Time.
25  #include <iostream.h>
26  #include "time1.h"
27  // Il costruttore di Time inizializza i dati membro a zero.
28  // Si assicura che tutti gli oggetti Time all'inizio siano
29  // in uno stato coerente.
30  Time::Time() { hour = minute = second = 0; }
31  // Imposta un nuovo valore per Time con il formato militare.
32  // Controlla la validità dei dati. Imposta a zero i valori non
33  // validi.
34  void Time::setTime( int h, int m, int s )
35  {
36     hour   = ( h >= 0 && h < 24 ) ? h : 0;
37     minute = ( m >= 0 && m < 60 ) ? m : 0;
38     second = ( s >= 0 && s < 60 ) ? s : 0;
39  }
40
41  // Print Time in military format
42  void Time::printMilitary()
43  {
44     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
45          << ( minute < 10 ? "0" : "" ) << minute;
46  }
47
48  // Visualizza l'orario in formato standard
49  void Time::printStandard()
50  {
51     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
52          << ":" << ( minute < 10 ? "0" : "" ) << minute
53          << ":" << ( second < 10 ? "0" : "" ) << second
54          << ( hour < 12 ? " AM" : " PM" );
55  }
```

Figura 6.5 Separazione dell'interfaccia e dell'implementazione per la classe Time (continua)

```
56  // Fig. 6.5: fig06_05.cpp
57  // Programma di prova per la classe Time1
58  // NOTA: Da compilare con time1.cpp
59  #include <iostream.h>
60  #include "time1.h"
61
62  // Programma di prova per testare la classe Time
63  int main()
64  {
65     Time t;  // istanzia l'oggetto t della classe time
66
67     cout << "The initial military time is ";
68     t.printMilitary();
69     cout << "\nThe initial standard time is ";
70     t.printStandard();
71
72     t.setTime( 13, 27, 6 );
73     cout << "\n\nMilitary time after setTime is ";
74     t.printMilitary();
75     cout << "\nStandard time after setTime is ";
76     t.printStandard();
77     // tentativo di impostare valori non validi
78     t.setTime( 99, 99, 99 );
79     cout << "\n\nAfter attempting invalid settings:\n"
80          << "Military time: ";
81     t.printMilitary();
82     cout << "\nStandard time: ";
83     t.printStandard();
84     cout << endl;
85     return 0;
86  }
```

```
The initial military time is 00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00
Standard time: 12:00:00 AM
```

Figura 6.5 Separazione dell'interfaccia e dell'implementazione per la classe Time.

Osservate che la dichiarazione della classe è racchiusa tra il codice destinato al preprocessore:

```
#ifndef TIME1_H
#define TIME1_H
...
#endif
```


Quando si crea un programma di notevoli dimensioni, nei file di intestazione vanno anche altre definizioni e dichiarazioni. Le direttive al preprocessore nelle figure evitano che il codice che si trova tra `#ifndef` e `#endif` sia incluso se è stato già definito il nome `TIME1_H`. Se l'intestazione non è stata inclusa precedentemente in un file, viene definito il nome `TIME1_H` tramite la direttiva `#define` e vengono incluse le istruzioni del file di intestazione. Se l'intestazione è già stato inclusa in precedenza, il nome `TIME1_H` è già stato definito, per cui il file di intestazione non viene nuovamente incluso. È tipico non accorgersi di includere più di una volta uno stesso file di intestazione, specie in progetti di grandi dimensioni dove gli stessi file di intestazione possono includerne altri. La convenzione in uso per i nomi delle costanti simboliche nelle direttive al preprocessore è che il nome del file di intestazione abbia un carattere di sottolineatura al posto del punto.

Collaudo e messa a punto 6.2

Utilizzate le direttive al preprocessore `#ifndef`, `#define` e `#endif` per evitare di includere più di una volta in un programma gli stessi file di intestazione.

Buona abitudine 6.2

Seguite la convenzione di utilizzare le direttive al preprocessore `#ifndef` e `#define` con il nome del file, sostituendo il punto con un carattere di sottolineatura.

6.8 Il controllo dell'accesso ai membri di una classe

Gli specificatori di accesso ai membri `public` e `private` (e `protected` che studieremo nel Capitolo 9) servono a controllare l'accesso ai dati e alle funzioni membro di una classe. La modalità di accesso di default dei membri di una classe è `private`, per cui tutti i membri elencati nell'intestazione di una classe all'inizio al primo specificatore esplicito sono automaticamente considerati `private`. Dopo ogni specificatore, la modalità indicata da esso si applica a tutti i membri che si trovano fino allo specificatore successivo o al termine della definizione della classe. Gli specificatori `public`, `private` e `protected` possono essere ripetuti, anche se ciò può generare confusione.

I membri `private` sono accessibili soltanto alle funzioni membro di tale classe (e le funzioni `friend`, come vedremo nel Capitolo 7). I membri `public` di una classe sono accessibili a tutte le funzioni del programma.

Lo scopo principale dei membri `public` è presentare ai client della classe una visione dei servizi offerti dalla classe; essi costituiscono l'interfaccia pubblica della classe. I client di una classe non devono preoccuparsi di come siano effettuati i vari compiti della classe e, dunque, i membri `private` non sono accessibili da essi. Questi componenti formano l'implementazione della classe.

Ingegneria del software 6.13

Il C++ permette di scrivere programmi indipendenti dall'implementazione. Se si modifica l'implementazione di una classe non occorre modificare il codice dei client, anche se questo probabilmente andrà ricompilato.

Errore tipico 6.6

Se una funzione che non è membro di una classe (né `friend` di tale classe) tenta di accedere ai membri `private` di quella classe, il compilatore segnalerà un errore di sintassi.

Il programma in Figura 6.6 mostra come i dati membro `private` siano accessibili soltanto tramite l'interfaccia pubblica della classe, ovvero tramite le funzioni membro `public`. In fase di compilazione si generano due errori perché si tenta di accedere a un dato membro che non è accessibile. Il programma in Figura 6.6 include `time1.h` ed è compilato con `time1.cpp` di Figura 6.5.

Buona abitudine 6.3

Anche se decidete di elencare per primi i membri `private` nella definizione di una classe, scrivete ugualmente la parola chiave `private`, anche se il compilatore la assume per default. Ciò contribuisce alla chiarezza del programma. La nostra preferenza personale è di elencare i membri `public` per primi, per evidenziare l'interfaccia della classe.

Buona abitudine 6.4

Anche se è possibile ripetere in qualsiasi ordine le etichette `public` e `private`, elencate prima tutti i membri `public` in un gruppo e poi tutti i membri `private` in un altro. L'attenzione di chi dovrà scrivere il client si focalizzerà così sull'interfaccia, piuttosto che sull'implementazione.

```
1   // Fig. 6.6: fig06_06.cpp
2   // Mostra gli errori che si verificano se si tenta di
3   // accedere a membri private della classe.
4   #include <iostream.h>
5   #include "time1.h"
6
7   int main()
8   {
9      Time t;
10
11     // Errore: 'Time::hour' non è accessibile
12     t.hour = 7;
13
14     // Errore: 'Time::minute' non è accessibile
15     cout << "minute = " << t.minute;
16
17     return 0;
18  }
```

```
Compiling FIG06_06.CPP:
Error FIG06_06.CPP 12: 'Time::hour' is not accessible
Error FIG06_06.CPP 15: 'Time::minute' is not accessible
```

Figura 6.6 Tentativo illecito di accedere ai dati private di una classe.

Ingegneria del software 6.14

I dati membro di una classe dovrebbero essere tutti `private`. Scrivete delle funzioni membro `public` per leggere e modificare il valore dei dati membro. Un'architettura di questo tipo serve a mantenere nascosta l'implementazione della classe.

Anche una funzione membro di un'altra classe o una funzione globale (cioè una funzione in vecchio stile C, non facente parte di nessuna classe) possono essere client di una determinata classe.

La modalità di default relativa all'accesso ai membri di una classe è `private`. La modalità può essere esplicitamente impostata a `public`, `protected` o `private`. La modalità di accesso di default per i membri di una `struct` è invece `public`. Tuttavia anche per le `struct` è possibile impostarla esplicitamente a `public`, `protected` o `private`.

Ingegneria del software 6.15

Chi progetta classi classifica i membri in `private`, `protected` e `public` per applicare il concetto di occultamento delle informazioni e il principio del minor privilegio.

Dire che un dato membro è `private` non equivale a dire che non può essere modificato da un client. I dati *possono* essere modificati tramite le funzioni membro o le funzioni `friend` di tale classe. Vedremo che queste funzioni devono essere progettate ponendo particolare attenzione a conservare l'integrità dei dati che vanno a manipolare.

L'accesso ai dati `private` di una classe dovrebbe essere controllato attentamente nelle funzioni membro, che si chiamano *funzioni di accesso* (o *metodi di accesso*). Per esempio, per leggere il valore di un dato `private`, la classe dovrebbe fornire una funzione *leggi*, o in inglese *get*. Ugualmente per modificare il dato sarebbe necessaria una funzione *modifica*, o *set*. A prima vista può sembrare che il fatto di poter modificare questi dati violi il concetto di dato privato.

Ma una funzione membro *set* può controllare i dati che l'utente vuole memorizzare nell'oggetto, per assicurare che abbiano sempre significato. Una funzione *set* può anche tradurre la forma dei dati utilizzata nell'interfaccia in quella utilizzata nell'implementazione, in quanto esse possono essere anche molto diverse. Una funzione *get* non deve necessariamente restituire i dati in un formato "grezzo" può invece effettuare un editing sui dati, limitando i dati visibili al client.

Ingegneria del software 6.16

Non occorre prevedere una coppia di funzioni `set`/`get` per ogni dato membro della classe: queste funzioni hanno un senso soltanto per i dati che è appropriato leggere o modificare.

Collaudo e messa a punto 6.3

Per agevolare la fase di debugging di un programma rendete `public` i dati e le funzioni membro delle classi: in questo modo sarà più semplice localizzare eventuali problemi relativi alla manipolazione dei dati e capire da quali funzioni dipendono.

6.9 Le funzioni di accesso e di utilità

Non tutte le funzioni membro devono necessariamente essere `public` per servire da interfaccia di una classe. Alcune possono restare `private` e diventare *funzioni di utilità* per le altre funzioni della classe.

Ingegneria del software 6.17

Le funzioni membro tendenzialmente appartengono a una di queste categorie: funzioni che leggono e restituiscono il valore di dati membro `private`; funzioni che impostano il valore di dati membro `private`; funzioni che implementano le operazioni caratteristiche della classe; funzioni che eseguono varie operazioni di base come l'inizializzazione degli oggetti, l'assegnamento, la conversione tra oggetti della classe e altri tipi di dato e la gestione della memoria per gli oggetti della classe.

Le funzioni di accesso possono leggere e visualizzare i dati. Un altro uso comune di queste funzioni è la verifica della verità o della falsità di una condizione: in tal caso si parla di *funzioni predicative*. Un esempio di funzione predicativa è `isEmpty` (in inglese: *è vuoto?*) per verificare se non ci sono elementi in una *classe container*, che è una classe in grado di contenere diversi oggetti. Esempi di classi container sono le liste concatenate, le pile e le code. Un programma può chiamare la funzione `isEmpty` prima di leggere un elemento dall'oggetto container. Allo stesso modo la funzione predicativa `isFull` (in inglese: *è pieno?*) può verificare se è rimasto ancora dello spazio in una classe container. Due utili funzioni predicative per la nostra classe `Time` potrebbero essere `isAM`, che verifica se l'orario è antimeridiano, e `isPM`, per verificare se l'orario è pomeridiano. Il programma in Figura 6.7 illustra il concetto di funzione di utilità. Una funzione di questo tipo non fa parte dell'interfaccia di una classe: possiamo dire piuttosto che si tratta di una funzione di supporto alle operazioni delle funzioni membro `public` di tale classe.Le funzioni di utilità non sono pensate per essere chiamate dai client.

```
1   // Fig. 6.7: salesp.h
2   // Definizione della classe SalesPerson
3   // Le funzioni membro sono definite in salesp.cpp
4   #ifndef SALESP_H
5   #define SALESP_H
6
7   class SalesPerson {
8   public:
9      SalesPerson();                  // costruttore
10     void getSalesFromUser();        // input dei ricavi mensili
11     void setSales( int, double ); // L'utente immette i
12                                     // ricavi di un mese.
13     void printAnnualSales();
14
15  private:
16     double totalAnnualSales();          // funzione di utilità
17     double sales[ 12 ];                 // i 12 ricavi mensili
18  };
19
20  #endif
```

Figura 6.7 Una funzione di utilità (continua)


```
21  // Fig. 6.7: salesp.cpp
22  // Funzioni membro per la classe SalesPerson
23  #include <iostream.h>
24  #include <iomanip.h>
25  #include "salesp.h"
26
27  // Il costruttore inizializza l'array
28  SalesPerson::SalesPerson()
29  {
30     for ( int i = 0; i < 12; i++ )
31        sales[ i ] = 0.0;
32  }
33
34  // Funzione per ottenere i 12 ricavi mensili dall'utente
35  // alla tastiera
36  void SalesPerson::getSalesFromUser()
37  {
38     double salesFigure;
39
40     for ( int i = 0; i < 12; i++ ) {
41        cout << "Enter sales amount for month "
42             << i + 1 << ": ";
43        cin >> salesFigure;
44        setSales( i, salesFigure );
45     }
46  }
47
48  // Funzione per impostare uno dei 12 ricavi mensili.
49  // Il valore del mese deve essere nell'intervallo 0   11.
50  void SalesPerson::setSales( int month, double amount )
51  {
52     if ( month >= 0 && month < 12 && amount > 0 )
53        sales[ month ] = amount;
54     else
55        cout << "Invalid month or sales figure" << endl;
56  }
57
58  // Visualizza il totale di vendita in un anno
59  void SalesPerson::printAnnualSales()
60  {
61     cout << setprecision( 2 )
62          << setiosflags( ios::fixed | ios::showpoint )
63          << "\nThe total annual sales are: $"
64          << totalAnnualSales() << endl;
65  }
66
67  // Funzione di utilità private per il calcolo del
68  // totale di vendita
```

Figura 6.7 Una funzione di utilità (continua)

```
69  double SalesPerson::totalAnnualSales()
70  {
71     double total = 0.0;
72
73     for ( int i = 0; i < 12; i++ )
74        total += sales[ i ];
75
76     return total;
77  }

78  // Fig. 6.7: fig06_07.cpp
79  // Mostra una funzione di utilità
80  // Da compilare con salesp.cpp
81  #include <iostream.h>
82  #include "salesp.h"
83
84  int main()
85  {
86     SalesPerson s;          // crea l'oggetto SalesPerson s
87
88     s.getSalesFromUser();   // notare il semplice codice
89                             // sequenziale
90     s.printAnnualSales();   // non ci sono strutture di
91                             // controllo in main
92     return 0;
93  }
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.58
```

Figura 6.7 Una funzione di utilità.

La classe `SalesPerson` contiene un array dei 12 ricavi di vendita mensili, inizializzato a zero dal costruttore e impostato poi con i valori immessi dall'utente dalla funzione `setSales`. La funzione membro pubblica `printAnnualSales` visualizza il totale delle vendite relativo agli ultimi 12 mesi. La funzione di utilità `totalAnnualSales` somma i 12 ricavi mensili, a beneficio di `printAnnualSales`. La funzione membro `printAnnualSales` effettua l'editing dei ricavi mensili, ponendoli nel formato di dollari.

Come notare, `main` include soltanto una semplice sequenza di chiamate di funzione: non c'è nessuna struttura di controllo.

Ingegneria del software 6.18

Uno degli aspetti della programmazione orientata agli oggetti è che, una volta definita la classe, la manipolazione degli oggetti richiede spesso soltanto una semplice sequenza di chiamate alle funzioni membro, e qualche o nessuna struttura di controllo. Al contrario le strutture di controllo sono molto comuni nell'implementazione delle funzioni membro della classe.

6.10 L'inizializzazione degli oggetti di una classe: i costruttori

Quando si crea un oggetto di una classe è possibile inizializzarne i membri tramite il costruttore della classe. Un costruttore è una funzione membro che ha lo stesso nome della classe. È compito del programmatore scrivere il costruttore, il quale verrà chiamato ogni volta che viene istanziato (creato) un nuovo oggetto della classe. I costruttori possono subire overloading per inizializzare gli oggetti della classe in vari modi. I dati membro possono essere inizializzati da subito tramite un costruttore o successivamente, magari tramite funzioni *set*, quando l'oggetto è stato già creato.

Errore tipico 6.7

I dati membro di una classe non possono essere inizializzati all'interno della definizione di una classe.

Errore tipico 6.8

I costruttori non possono restituire alcun valore. Se ne specificate uno, o se tentate di restituirne uno dal suo interno, commettete un errore di sintassi.

Buona abitudine 6.5

Quando è appropriato (cioè quasi sempre) scrivete un costruttore, per essere sicuri che ogni oggetti sia inizializzato con dei valori significativi. In particolare i dati membro puntatore dovrebbero essere inizializzati con un valore corretto o a 0.

Collaudo e messa a punto 6.4

Ogni funzione membro (e `friend`) che modifica i dati `private` di un oggetto dovrebbe verificare se i dati scritti sono significativi e se lasciano quindi l'oggetto in uno stato coerente.

Nella dichiarazione di un oggetto di una classe è possibile racchiudere tra parentesi gli inizializzatori a destra del nome dell'oggetto e prima del punto e virgola. Questi inizializzatori sono passati poi come argomenti al costruttore della classe. Esamineremo tra breve alcuni esempi di queste chiamate ai costruttori. Notate quindi che i programmatori non chiamano esplicitamente i costruttori, ma possono passare loro dei valori.

6.11 I costruttori e gli argomenti di default

Il costruttore del file `time1.cpp` (Figura 6.5) inizializzava `hour`, `minute` e `second` a `0`, ovvero a mezzanotte nel formato militare. I costruttori possono contenere argomenti di default. La Figura 6.8 ridefinisce il costruttore di `Time` includendo degli zero come argomenti di default per ciascuna delle variabili. Grazie agli argomenti di default, anche se la chiamata al costruttore non contiene argomenti, si è sicuri che l'oggetto sia inizializzato in uno stato coerente. Se il programmatore scrive un costruttore con argomenti di default, questo diventerà anche il *costruttore di default*, ovvero il costruttore che sarà chiamato quando non sono specificati inizializzatori. È possibile al massimo un costruttore di default per classe.

```
1  // Fig. 6.8: time2.h
2  // Dichiarazione della classe Time.
3  // Le funzioni membro sono definite in time2.cpp
4
5  // direttive al preprocessore per evitare
6  // che il file di intestazione sia incluso più di una volta
7  #ifndef TIME2_H
8  #define TIME2_H
9
10 // Definizione di Time come tipo di dato astratto
11 class Time {
12 public:
13    Time(int=0, int=0, int=0); //costruttore di default
14    void setTime(int,int,int); //imposta hour,minute,second
15    void printMilitary(); //mostra l'ora in formato militare
16    void printStandard(); //mostra l'ora in formato standard
17 private:
18    int hour;     // 0   23
19    int minute;   // 0   59
20    int second;   // 0   59
21 };
22
23 #endif

24 // Fig. 6.8: time2.cpp
25 // Definizioni delle funzioni membro di Time.
26 #include <iostream.h>
27 #include "time2.h"
28 // Il costruttore di Time inizializza ogni dato membro
29 // a zero. In questo modo tutti gli oggetti Time si
30 // troveranno all'inizio in uno stato coerente.
31 Time::Time( int hr, int min, int sec )
32    { setTime( hr, min, sec ); }
33 // Imposta un nuovo valore per Time con il formato
34 // militare. Controlla la validità sui dati.
35 // Imposta a zero i dati non validi.
```

Figura 6.8 Un costruttore con argomenti di default (continua)


```
36 void Time::setTime( int h, int m, int s )
37 {
38    hour   = ( h >= 0 && h < 24 ) ? h : 0;
39    minute = ( m >= 0 && m < 60 ) ? m : 0;
40    second = ( s >= 0 && s < 60 ) ? s : 0;
41 }
42
43 // Visualizza Time in formato militare
44 void Time::printMilitary()
45 {
46    cout << ( hour < 10 ? "0"   "" ) << hour << ":"
47         << ( minute < 10 ? "0" : "" ) << minute;
48 }
49
50 // Visualizza Time in formato standard
51 void Time::printStandard()
52 {
53    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
54         << ":" << ( minute < 10 ? "0"   "" ) << minute
55         << ":" << ( second < 10 ? "0"   "" ) << second
56         << ( hour < 12 ? " AM" : " PM" );
57 }

58 // Fig. 6.8: fig06_08.cpp
59 // Mostra un costruttore di default
60 // per la classe Time.
61 #include <iostream.h>
62 #include "time2.h"
63
64 int main()
65 {
66    Time t1,                    // tutti argomenti di default
67         t2(2),                 // minute e second di default
68         t3(21, 34),            // second di default
69         t4(12, 25, 42),        // tutti i valori specificati
70         t5(27, 74, 99);        // valori non validi specificati
71
72    cout << "Constructed with:\n"
73         << "all arguments defaulted:\n   ";
74    t1.printMilitary();
75    cout << "\n   ";
76    t1.printStandard();
77
78    cout << "\nhour specified; minute and second defaulted:"
79         << "\n   ";
80    t2.printMilitary();
81    cout << "\n   ";
82    t2.printStandard();
83
```

Figura 6.8 Un costruttore con argomenti di default (continua)

```
84     cout << "\nhour and minute specified; second defaulted:"
85          << "\n   ";
86     t3.printMilitary();
87     cout << "\n   ";
88     t3.printStandard();
89
90     cout << "\nhour, minute, and second specified:"
91          << "\n   ";
92     t4.printMilitary();
93     cout << "\n   ";
94     t4.printStandard();
95
96     cout << "\nall invalid values specified:"
97          << "\n   ";
98     t5.printMilitary();
99     cout << "\n   ";
100    t5.printStandard();
101    cout << endl;
102
103    return 0;
104 }
```

```
Constructed with:
all arguments defaulted:
   00:00
   12:00:00 AM
hour specified; minute and second defaulted:
   02:00
   2:00:00 AM
hour and minute specified; second defaulted:
   21:34
   9:34:00 PM
hour, minute, and second specified:
   12:25
   12:25:42 PM
all invalid values specified:
   00:00
   12:00:00 AM
```

Figura 6.8 Un costruttore con argomenti di default.

In questo programma il costruttore chiama la funzione membro `setTime` con i valori di default per assicurarsi che il valore fornito per `hour` si trovi nell'intervallo 0-23, e che i valori per `minute` e `second` si trovano nell'intervallo 0-59. Se uno dei valori non è accettabile, viene azzerato da `setTime`: questo è un esempio di verifica della consistenza dei dati.

Notate che possiamo scrivere il costruttore di `Time` includendo le stesse istruzioni della funzione membro `setTime`. Ciò migliora l'efficienza del codice, perché la chiamata extra a `setTime` viene eliminata. Tuttavia scrivendo in modo identico il codice del costruttore di `Time` e della funzione `setTime` complichiamo la manutenzione del programma. Infatti,

se modifichiamo l'implementazione di `setTime` dobbiamo poi adeguare quella del costruttore. Invece se il costruttore chiama direttamente `setTime`, possiamo modificare come desideriamo l'implementazione di `setTime` senza dover modificare anche sul costruttore. In questo modo diminuiamo anche la probabilità di errori. Se vogliamo ottimizzare il costruttore possiamo dichiararlo esplicitamente come funzione `inline` o possiamo definirlo nella definizione di classe, il che lo rende `inline` automaticamente.

Ingegneria del software 6.19

Se c'è già una funzione della classe che fornisce le stesse funzionalità che dovrebbe offrire il costruttore (o un'altra funzione membro), invocate quella funzione dal costruttore (o dalla funzione membro). Ciò semplifica la manutenzione del codice e riduce la probabilità di errori, nel caso l'implementazione venga modificata in un secondo tempo. Come regola generale, evitate di ripetere intere porzioni di codice.

Buona abitudine 6.6

Dichiarate i valori di default per gli argomenti solo nei prototipi di funzione che si trovano nella definizione di classe nel file di intestazione.

Errore tipico 6.9

Non specificate gli inizializzatori di default per una stessa funzione membro sia in un file di intestazione che nella definizione della funzione o commettete un errore.

Il programma in Figura 6.8 inizializza cinque oggetti `Time`: uno con tutti e tre gli argomenti di default, un altro con un solo argomento specificato, uno con due argomenti specificati, uno con tutti e tre gli argomenti specificati e un ultimo con i tre argomenti specificati non validi. I dati membro di ogni oggetto sono visualizzati dopo l'istanza e l'inizializzazione.

Se non si definisce alcun costruttore per una classe, il compilatore ne crea uno di default. Un costruttore di questo tipo non effettua alcuna inizializzazione, per cui quando l'oggetto viene creato non è certo che si trovi in uno stato coerente.

Ingegneria del software 6.20

Una classe può anche essere priva di un costruttore di default.

6.12 I distruttori

Un *distruttore* è una funzione membro speciale di una classe. Il nome del distruttore di una classe è formato dal carattere tilde (~) seguito dal nome della classe. Questa convenzione ha una ragione intuitiva, perché come vedremo l'operatore tilde è l'operatore di complemento sui bit, e in un certo senso il distruttore è il complemento del costruttore.

Il distruttore di una classe viene invocato quando l'oggetto è distrutto, cioè quando l'esecuzione del programma oltrepassa lo scope in cui l'oggetto è istanziato. Il distruttore non distrugge realmente l'oggetto, ma effettua delle operazioni di base sulla memoria occupata dall'oggetto prima di restituirla al sistema, in modo che possa essere riutilizzata per memorizzare nuovi oggetti. Un distruttore non riceve parametri e non restituisce alcun valore. Una classe può avere un solo distruttore e non è consentito l'overloading.

Errore tipico 6.10

Commettete un errore di sintassi se passate degli argomenti a un distruttore, se ne specificate il tipo di dato restituito, se restituite un valore dall'interno di esso o se ne effettuate l'overloading.

Tra breve vedremo alcuni esempi di classi che contengono distruttori molto utili. Nel Capitolo 8 vedremo come i distruttori siano indispensabili nel caso di oggetti dinamici, come array e stringhe. Nel Capitolo 7 parleremo di come allocare e deallocare dinamicamente dello spazio in memoria.

Ingegneria del software 6.21

Andando avanti nel nostro corso vedremo come i costruttori e i distruttori siano molto più importanti di quanto si possa evincere dalla nostra breve introduzione.

6.13 Quando sono chiamati i costruttori e i distruttori?

Le chiamate a costruttori e distruttori sono automatiche. Il loro ordine dipende dall'ordine in cui il flusso dell'esecuzione entra ed esce dallo scope in cui sono istanziati gli oggetti. In genere i distruttori sono chiamati in ordine inverso rispetto ai costruttori. Ad ogni modo, la categoria di memorizzazione di un oggetto può alterare l'ordine di chiamata dei distruttori, e lo vedremo nel programma in Figura 6.9.

I costruttori degli oggetti che hanno scope globale sono chiamati prima di ogni altra funzione (`main` inclusa), ma se gli oggetti globali sono sparsi su file diversi l'ordine di chiamata dei costruttori non è predeterminabile. I distruttori corrispondenti sono chiamati quando `main` termina o quando viene chiamata la funzione `exit` che, invocata in qualunque punto di un programma termina immediatamente la sua esecuzione.

I costruttori sono chiamati per gli oggetti locali automatici nel momento in cui l'esecuzione raggiunge il punto in cui sono definiti. I distruttori corrispondenti sono chiamati quando l'esecuzione oltrepassa lo scope degli oggetti in questione, ovvero al termine del blocco in cui sono stati definiti. I costruttori e i distruttori degli oggetti automatici sono chiamati a ogni entrata e uscita dal rispettivo scope.

Per quanto riguarda gli oggetti locali `static`, i costruttori sono chiamati soltanto una volta, e cioè la prima volta che l'esecuzione raggiunge il punto in cui sono definiti. I distruttori corrispondenti sono chiamati quando `main` termina o quando viene chiamata la funzione `exit`.

Il programma in Figura 6.9 illustra l'ordine in cui sono chiamati i costruttori e i distruttori degli oggetti `CreateAndDestroy`, in diversi livelli di visibilità. Il programma definisce `first` nello scope globale. Il suo costruttore è chiamato all'inizio e il suo distruttore è chiamato al termine dell'esecuzione, dopo la distruzione di tutti gli altri oggetti.


```
1   // Fig. 6.9: create.h
2   // Definizione della classe CreateAndDestroy.
3   // Funzioni membro definite in create.cpp.
4   #ifndef CREATE_H
5   #define CREATE_H
6
7   class CreateAndDestroy {
8   public:
9      CreateAndDestroy( int );  // costruttore
10     ~CreateAndDestroy();      // distruttore
11  private:
12     int data;
13  };
14
15  #endif

16  // Fig. 6.9: create.cpp   Definizioni delle funzioni
17  // funzioni membro della classe CreateAndDestroy
18  #include <iostream.h>
19  #include "create.h"
20
21  CreateAndDestroy::CreateAndDestroy( int value )
22  {
23     data = value;
24     cout << "Object " << data << "   constructor";
25  }
26
27  CreateAndDestroy::~CreateAndDestroy()
28     { cout << "Object " << data << " destructor " << endl; }

29  // Fig. 6.9: fig06_09.cpp
30  // Mostra l'ordine di chiamata di
31  // costruttori e distruttori.
32  #include <iostream.h>
33  #include "create.h"
34
35  void create( void );   // prototipo
36
37  CreateAndDestroy first( 1 );  // oggetto globale
38
39  int main()
40  {
41     cout << "   (global created before main)" << endl;
42
43     CreateAndDestroy second( 2 );        // local object
44     cout << "   (local automatic in main)" << endl;
45
```

Figura 6.9 Programma che illustra l'ordine di chiamata di costruttori e distruttori (continua)

```
46      static CreateAndDestroy third( 3 );  // local object
47      cout << "   (local static in main)" << endl;
48
49      create();  // call function to create objects
50
51      CreateAndDestroy fourth( 4 );        // local object
52      cout << "   (local automatic in main)" << endl;
53      return 0;
54   }
55
56   // Funzione per creare oggetti
57   void create( void )
58   {
59      CreateAndDestroy fifth( 5 );
60      cout << "   (local automatic in create)" << endl;
61
62      static CreateAndDestroy sixth( 6 );
63      cout << "   (local static in create)" << endl;
64
65      CreateAndDestroy seventh( 7 );
66      cout << "   (local automatic in create)" << endl;
67   }
```

```
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Figura 6.9 Programma che illustra l'ordine di chiamata di costruttori e distruttori.

La funzione `main` dichiara tre oggetti: `second` e `fourth` sono oggetti locali automatici, mentre `third` è un oggetto locale statico. I costruttori di tutti questi oggetti sono chiamati quando l'esecuzione raggiunge il punto in cui sono dichiarati. I distruttori di `fourth` e `second` sono chiamati in quest'ordine, al termine di `main`. Siccome `third` è un oggetto `static`, esiste fino al termine del programma: il suo distruttore è chiamato prima di quello di `first`, ma dopo quelli di tutti gli altri oggetti.

Anche la funzione `create` dichiara tre oggetti: `fifth` e `seventh` sono oggetti locali automatici, mentre `sixth` è un oggetto locale `static`. I distruttori di `seventh` e `fifth` sono chiamati nell'ordine, quando viene raggiunta la fine della funzione `create`. L'oggetto `sixth` è `static`, perciò esiste fino al termine del programma: il suo distruttore è chiamato prima di quelli di `third` e `first`, ma dopo quelli di tutti gli altri oggetti.

6.14 L'utilizzo dei dati e delle funzioni membro

I dati membro `private` di una classe possono essere manipolati soltanto dalle funzioni membro (e `friend`) della classe. Un esempio di manipolazione può essere l'aggiornamento del bilancio di un conto cliente, tramite il dato membro `BankAccount`, per mezzo della *funzione* `computeInterest`.

Di solito le classi forniscono funzioni membro `public` che consentono ai client di effettuare la modifica (*set*) o la lettura (*get*) dei dati `private`. Queste funzioni non si devono chiamare obbligatoriamente *set* e *get*, ma si tratta di una convenzione abbastanza diffusa. Per esempio la funzione membro che modifica il dato membro `interestRate` si chiamerà tipicamente `setInterestRate`, mentre quella che legge `interestRate` si chiamerà `getInterestRate`. Le funzioni *get* si chiamano anche funzioni di *interrogazione* o di *query*.

Potreste pensare che la presenza di funzioni *set* e *get* equivalga in sostanza a rendere `public` i dati membro. In realtà questa è una delle sottigliezze logiche del C++ che contribuiscono a rendere questo linguaggio ancora una volta apprezzabile dal punto di vista dello sviluppo di progetti complessi. Se un dato membro è `public`, può essere letto e modificato a piacimento da qualsiasi altra funzione del programma. Se esso è `private`, è vero che le altre funzioni del programma possono accedere in qualche modo al suo valore, ma questo valore viene restituito filtrato dalla funzione *get*, che controllerà la formattazione e la visualizzazione del dato. Una funzione *set* effettuerà un controllo attento prima di scrivere un valore nel dato e quindi di modificarne il contenuto. In questo modo si può contare sul fatto che il valore del dato sia sempre significativo. I valori incoerenti saranno respinti dalle funzioni *set*: ciò vale, per esempio, se si tenta di impostare il giorno del mese a 37, il peso di una persona a un valore negativo, una quantità prettamente numerica a un valore alfanumerico, l'esito in centesimi di un esame a 185, e così via.

Ingegneria del software 6.22

Per assicurare l'integrità dei dati nei vostri programmi, rendete i dati membro `private` *e controllatene l'accesso, soprattutto in scrittura, tramite funzioni membro* `public`.

Collaudo e messa a punto 6.5

Rendere i dati membro `private` *non significa assicurarne automaticamente l'integrità: occorre prevedere dei meccanismi di convalida dei valori. La struttura del C++, comunque, agevola la progettazione di questi controlli.*

Buona abitudine 6.7

Le funzioni membro `set`, *che scrivono nuovi valori nei dati membro, dovrebbero verificare che tali valori siano accettabili: se non lo sono, le funzioni* `set` *dovrebbero tentare di conservare la coerenza del dato membro che vanno a modificare e dell'oggetto in generale.*

Un client di una classe dovrebbe essere avvisato quando cerca di assegnare a un dato membro un valore non accettabile. Di solito le funzioni *set* restituiscono un valore che indica se l'assegnamento è riuscito oppure no. Questo consente a un client di effettuare un test sul valore restituito dalla funzione *set* e di determinare quindi se l'oggetto che sta manipolando è valido e, in caso contrario, può scegliere di intraprendere un'azione determinata.

Il programma in Figura 6.10 estende la precedente classe `Time`, includendo le funzioni *get* e *set* per i dati membro `hour`, `minute` e `second`. Le funzioni *set* controllano molto attentamente i valori ricevuti prima di scriverli nell'oggetto: se il client tenta di modificare in modo scorretto un dato membro quest'ultimo sarà sempre azzerato, lasciando sempre l'oggetto in uno stato coerente. Ogni funzione *get* restituisce semplicemente il valore del dato membro corrispondente. Il programma si serve prima delle funzioni *set* per scrivere dei valori validi nei dati membro `private` dell'oggetto `t`, e poi delle funzioni *get* per leggere i valori e inviarli in output. Successivamente le funzioni *set* tentano di scrivere valori non validi in `hour` e `second` e un valore valido in `minute`, e poi le funzioni *get* leggono i valori e li inviano in output. L'output conferma che i valori non validi hanno causato l'azzeramento dei dati membro corrispondenti. Infine il programma imposta l'orario a `11:58:00` e incrementa il valore di `minute` di 3 con una chiamata alla funzione `incrementMinutes`. Questa funzione non è membro della classe, e quindi non può che utilizzare le funzioni *get* e *set* per alterare il dato membro `minute`. Questo meccanismo funziona, ma appesantisce il programma con una serie di chiamate di funzione. Nel prossimo capitolo illustreremo il concetto di funzione `friend` che elimina questo problema di efficienza.

Errore tipico 6.11

Un costruttore può chiamare altre funzioni membro della sua classe, come le funzioni `set` *e* `get`*, ma dato che l'oggetto non è stato ancora inizializzato i dati membro possono non essere in uno stato coerente. Questo significa che i dati membro sono utilizzati prima di essere inizializzati, il che porta facilmente a commettere errori logici.*

```
1  // Fig. 6.10: time3.h
2  // Dichiarazione della classe Time.
3  // le funzioni membro sono definite in time3.cpp
4
5  // direttive al preprocessore per evitare di
6  // includere il file di intestazione più di una volta
7  #ifndef TIME3_H
8  #define TIME3_H
9
10 class Time {
11 public:
12    Time( int = 0, int = 0, int = 0 );  // costruttore
13
14    // set functions
15    void setTime(int,int,int); //i mposta hour,minute,second
16    void setHour( int );    // imposta hour
17    void setMinute( int ); // imposta minute
```

Figura 6.10 Utilizzo delle funzioni set e get (continua)


```
18    void setSecond( int ); // imposta second
19
20    // funzioni get
21    int getHour();          // restituisce hour
22    int getMinute();        // restituisce minute
23    int getSecond();        // restituisce second
24
25    void printMilitary();  // output in formato militare
26    void printStandard();  // output in formato standard
27
28 private:
29    int hour;              // 0   23
30    int minute;            // 0   59
31    int second;            // 0   59
32 };
33
34 #endif

35 // Fig. 6.10: time3.cpp
36 // Definizioni delle funzioni membro della classe Time.
37 #include "time3.h"
38 #include <iostream.h>
39 // Costruttore per inizializzare i dati private.
40 // Chiama la funzione membro setTime per impostare le
41 // variabili. I valori di default sono 0 (cfr. la
42 // definizione di class).
43 Time::Time( int hr, int min, int sec )
44    { setTime( hr, min, sec ); }
45
46 // Imposta i valori di hour, minute e second.
47 void Time::setTime( int h, int m, int s )
48 {
49    setHour( h );
50    setMinute( m );
51    setSecond( s );
52 }
53
54 // Imposta il valore di hour
55 void Time::setHour( int h )
56    { hour = ( h >= 0 && h < 24 ) ? h : 0; }
57
58 // Imposta il valore di minute
59 void Time::setMinute( int m )
60    { minute = ( m >= 0 && m < 60 ) ? m : 0; }
61
62 // Imposta il valore di second
63 void Time::setSecond( int s )
64    { second = ( s >= 0 && s < 60 ) ? s : 0; }
```

Figura 6.10 Utilizzo delle funzioni set e get (continua)

```
65
66  // Restituisce il valore di hour
67  int Time::getHour() { return hour; }
68
69  // Restituisce il valore di minute
70  int Time::getMinute() { return minute; }
71
72  // Restituisce il valore di second
73  int Time::getSecond() { return second; }
74
75  // Visualizza l'orario in formato militare
76  void Time::printMilitary()
77  {
78     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
79          << ( minute < 10 ? "0" : "" ) << minute;
80  }
81
82  // Visualizza l'orario in formato standard
83  void Time::printStandard()
84  {
85     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
86          << ":" << ( minute < 10 ? "0" : "" ) << minute
87          << ":" << ( second < 10 ? "0" : "" ) << second
88          << ( hour < 12 ? " AM" : " PM" );
89  }
```

```
90  // Fig. 6.10: fig06_10.cpp
91  // Mostra le funzioni set e get della classe Time
92  #include <iostream.h>
93  #include "time3.h"
94
95  void incrementMinutes( Time &, const int );
96
97  int main()
98  {
99     Time t;
100
101    t.setHour( 17 );
102    t.setMinute( 34 );
103    t.setSecond( 25 );
104
105    cout << "Result of setting all valid values:\n"
106         << "  Hour: " << t.getHour()
107         << "  Minute: " << t.getMinute()
108         << "  Second: " << t.getSecond();
109
110    t.setHour( 234 );    // hour non valido azzerato
111    t.setMinute( 43 );
```

Figura 6.10 Utilizzo delle funzioni set e get (continua)


```
112    t.setSecond( 6373 ); // second non valido azzerato
113
114    cout << "\n\nResult of attempting to set invalid
115            hour and"
116         << " second:\n  Hour: " << t.getHour()
117         << "  Minute: " << t.getMinute()
118         << "  Second: " << t.getSecond() << "\n\n";
119
120    t.setTime( 11, 58, 0 );
121    incrementMinutes( t, 3 );
122
123    return 0;
124 }
125
126 void incrementMinutes(Time &tt, const int count)
127 {
128    cout << "Incrementing minute " << count
129         << " times:\nStart time: ";
130    tt.printStandard();
131
132    for ( int i = 0; i < count; i++ ) {
133       tt.setMinute( ( tt.getMinute() + 1 ) % 60);
134
135       if ( tt.getMinute() == 0 )
136          tt.setHour( ( tt.getHour() + 1 ) % 24);
137
138       cout << "\nminute + 1: ";
139       tt.printStandard();
140    }
141
142    cout << endl;
143 }
```

```
Result of setting all valid values:
  Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid hour and second:
  Hour: 0  Minute: 43  Second: 0

Incrementing minute 3 times:
Start time: 11:58:00 AM
minute + 1: 11:59:00 AM
minute + 1: 12:00:00 PM
minute + 1: 12:01:00 PM
```

Figura 6.10 Utilizzo delle funzioni set e get.

Le funzioni *set* giocano un ruolo importante nella metodologia di programmazione perché effettuano controlli di validità sui dati. Inoltre le funzioni *set* e *get* svolgono un'altra funzione importante.

Ingegneria del software 6.23

Se accedete ai dati `private` di una classe tramite le funzioni `set` e `get`, non solo proteggete i dati da valori non validi, ma isolate l'implementazione della classe, separandola dall'interfaccia. Ciò significa che potrete modificare l'implementazione dei dati, ad esempio per ridurre la memoria occupata o per migliorare l'efficienza delle loro manipolazioni, riscrivendo soltanto le funzioni membro: i client non devono essere modificati, sempre che abbiate lasciato inalterata l'interfaccia della classe. In ogni caso, probabilmente, i client andranno ricompilati.

6.15 Un sottile errore logico: restituire un riferimento a un dato membro private

Un riferimento a un oggetto è un alias per il *nome* dell'oggetto, e può essere quindi utilizzata come *lvalue*, cioè nella porzione sinistra di un assegnamento. Sfortunatamente questa funzionalità può essere usata in un modo improprio, restituendo dall'interno di una funzione membro `public` un riferimento non costante a un dato membro `private` della classe. Il programma in Figura 6.11 utilizza una versione semplificata della classe `Time` per illustrare questo tipo di problema. Il dato restituito rende la chiamata alla funzione `badSetHour` un alias per il dato membro privato `hour`! Di conseguenza la chiamata alla funzione può essere utilizzata in tutti i casi in cui si potrebbe utilizzare il dato membro `private`, persino come *lvalue* in un assegnamento!

Buona abitudine 6.8

Una funzione membro `public` non deve restituire un riferimento non costante (o un puntatore) a un dato membro `private`, altrimenti si viola l'incapsulamento della classe.

```
1   // Fig. 6.11: time4.h
2   // Dichiarazione della classe Time.
3   // Le funzioni membro sono definite in time4.cpp
4   // direttive al preprocessore per evitare di
5   // includere il file di intestazione più di una volta
6   #ifndef TIME4_H
7   #define TIME4_H
8
9   class Time {
10  public:
11     Time( int = 0, int = 0, int = 0 );
12     void setTime( int, int, int );
13     int getHour();
14     int &badSetHour( int );  // restituisce un riferimento:
15     // PERICOLOSO!
16  private:
17     int hour;
18     int minute;
```

Figura 6.11 Una trappola di programmazione: restituire un riferimento a un dato private (continua)


```
19     int second;
20  };
21  #endif

22  // Fig. 6.11: time4.cpp
23  // Definizioni delle funzioni membro della classe Time.
24  #include "time4.h"
25  #include <iostream.h>
26  // Costruttore per inizializzare i dati private.
27  // Chiama la funzione setTime per impostare le variabili.
28  // I valori di default sono 0 (cfr. la definizione di classe).
29  Time::Time( int hr, int min, int sec )
30     { setTime( hr, min, sec ); }
31  // Imposta il valore di hour, minute e second.
32  void Time::setTime( int h, int m, int s )
33  {
34     hour   = ( h >= 0 && h < 24 ) ? h : 0;
35     minute = ( m >= 0 && m < 60 ) ? m : 0;
36     second = ( s >= 0 && s < 60 ) ? s : 0;
37  }
38
39  // Legge il valore di hour
40  int Time::getHour() { return hour; }
41
42  // CATTIVA TECNICA DI PROGRAMMAZIONE:
43  // Restituire un riferimento a un dato membro private.
44  int &Time::badSetHour( int hh )
45  {
46     hour = ( hh >= 0 && hh < 24 ) ? hh   0;
47
48     return hour;  // restituisce un riferimento: PERICOLOSO!
49  }
50  // Fig. 6.11: fig06_11.cpp
51  // Mostra una funzione membro public che
52  // restituisce un riferimento a un dato membro private.
53  // La classe Time è stata ripensata per questo esempio.
54  #include <iostream.h>
55  #include "time4.h"
56
57  int main()
58  {
59     Time t;
60     int &hourRef = t.badSetHour( 20 );
61
62     cout << "Hour before modification: " << hourRef;
63     hourRef = 30;  // modification with invalid value
64     cout << "\nHour after modification: " << t.getHour();
```

Figura 6.11 Una trappola di programmazione: restituire un riferimento a un dato private (continua)

```
65       // Pericolo: Chiamata di funzione che restituisce
66       // un riferimento che può essere utilizzata come lvalue!
67       t.badSetHour(12) = 74;
68       cout << "\n\n********************************\n"
69            << "POOR PROGRAMMING PRACTICE!!!!!!!!!\n"
70            << "badSetHour as an lvalue, Hour: "
71            << t.getHour()
72            << "\n********************************" << endl;
73
74       return 0;
75    }
```

```
Hour before modification: 20
Hour after modification: 30

********************************
POOR PROGRAMMING PRACTICE!!!!!!!!!
badSetHour as an lvalue, Hour: 74
********************************
```

Figura 6.11 Una trappola di programmazione: restituire un riferimento a un dato **private**.

Il programma dichiara `t` come oggetto di tipo `Time`, e il riferimento `hourRef` che riceve il valore restituito dalla chiamata a `t.badSetHour(20)`, un'altro riferimento.

Successivamente il programma visualizza il valore dell'alias `hourRef`, quindi utilizza l'alias per impostare a `30` il valore di `hour` (valore non valido) e visualizza nuovamente il valore. Infine, la stessa chiamata di funzione è utilizzata come *lvalue*, in quanto le viene assegnato il valore `74` (ancora un valore non valido), e il valore viene visualizzato un'ultima volta.

6.16 L'assegnamento tra oggetti: la copia di default membro a membro

L'operatore di assegnamento (=) può essere utilizzato per assegnare un oggetto a un altro dello stesso tipo. Per default questa operazione viene effettuata copiando individualmente *ogni membro di un oggetto in quello corrispondente dell'altro, effettuando la cosiddetta copia di default membro a membro* (cfr. Figura 6.12).

Vedremo in seguito che la copia membro a membro può generare grossi problemi se copia dati membro che riferiscono aree di memoria allocate dinamicamente: nel Capitolo 8 torneremo a parlare di questo problema e ne cercheremo la soluzione.

Le funzioni possono ricevere un oggetto come argomento e possono restituirne uno. Per default entrambe le operazioni sono effettuate per valore, cioè la funzione riceve o restituisce una copia dell'oggetto (presenteremo numerosi esempio nel Capitolo 8).

Obiettivo efficienza 6.4

Passare un oggetto per valore è una cosa positiva dal punto di vista della sicurezza perché la funzione chiamata non può accedere all'oggetto originario, ma ha un costo sull'efficienza nel caso di grossi oggetti, perché ne viene effettuata prima una copia in memoria. Un oggetto può essere passato per riferimento tramite un puntatore o un riferimento. La chiamata per riferimento è efficiente ma debole dal punto di vista della sicurezza perché la funzione accede all'oggetto originario. La chiamata per riferimento costante costituisce una valida alternativa sia per la sicurezza che per l'efficienza.

```
1    // Fig. 6.12: fig06_12.cpp - Mostra come sia
2    // possibile l'assegnamento tra oggetti di una stessa
3    // classe tramite alla copia membro a membro (default)
4    #include <iostream.h>
5
6    // Simple Date class
7
8    class Date {
9    public:
10      Date(int=1, int=1, int=1990); // costruttore di default
11      void print();
12   private:
13      int month;
14      int day;
15      int year;
16   };
17
18   // Semplice costruttore di Date senza controlli di validità
19   Date::Date( int m, int d, int y )
20   {
21      month = m;
22      day = d;
23      year = y;
24   }
25
26   // Visualizza Date nella forma mm-dd-yyyy
27   void Date::print()
28      { cout << month << '-' << day << '-' << year; }
29
30   int main()
31   {
32      Date date1(7,4,1993),date2; //d2 per default vale 1/1/90
33
34      cout << "date1 = ";
35      date1.print();
36
37      cout << "\ndate2 = ";
38      date2.print();
```

Figura 6.12 Assegnamento di due oggetti con la copia di default membro a membro (continua)

```
38
39       date2 = date1;    // assegnamento
40       cout << "\n\nAfter default memberwise copy, date2 = ";
41       date2.print();
42
43       cout << endl;
44
45       return 0;
46    }
```

```
date1 = 7-4-1993
date2 = 1-1-1990

After default memberwise copy, date2 = 7-4-1993
```

Figura 6.12 Assegnamento di due oggetti con la copia di default membro a membro.

6.17 Ancora sul concetto di software riutilizzabile

I programmatori che utilizzano il C++ concentrano i loro sforzi per creare classi utili a tutti. L'opportunità commerciale di catalogare e distribuire classi che effettuano i compiti più diversi è enorme. In tutto il mondo ne circolano già moltissime, e molte altre sono in fase di sviluppo con l'intento di renderle accessibili a tutti. I software di oggi si creano soprattutto a partire da componenti già disponibili, progettati e verificati con estrema attenzione, ben documentati e portabili. Ciò comporta una drastica riduzione dei tempi nello sviluppo di software complessi e di alta qualità ed ha dato luogo alla nascita di nuovi strumenti per lo sviluppo di applicazioni che vanno sotto il nome di *RAD* (*Rapid Application Development* o, in italiano, *sviluppo rapido di applicazioni*).

Prima di poter sfruttare appieno le potenzialità del software riutilizzabile ci sono ancora alcuni problemi significativi da risolvere. C'è bisogno di schemi di licenza e di protezione, in modo che le copie originali delle classi non possano essere alterate, di schemi descrittivi, in modo che i progettisti di nuovi sistemi possano determinare se gli oggetti già esistenti possono soddisfare le loro esigenze, di meccanismi di catalogazione, in modo che i progettisti possano sfogliare l'enciclopedia delle classi esistenti e scegliere quelle che si avvicinano di più ai loro bisogni. Questi problemi sono ancora *in fase di studio anche* negli ambienti di ricerca, e possiamo dirvi che le motivazioni nel cercare delle soluzioni sono enormi, al pari del potenziale valore delle soluzioni.

6.18 Pensare in termini di oggetti: programmazione delle classi del simulatore [progetto opzionale]

Nelle sezioni "Pensare in termini di oggetti" dei precedenti capitoli abbiamo introdotto le nozioni fondamentali della progettazione orientata agli oggetti e le abbiamo applicate al nostro simulatore di ascensore. Nel corso di questo capitolo abbiamo parlato dei dettagli di programmazione delle classi in C++, e quindi siamo in grado di iniziare l'implementazione

del nostro progetto. In questa sezione utilizzeremo il diagramma delle classi UML per individuare i file di intestazione che conterranno la definizione delle classi.

6.18.1 Implementazione: visibilità

In questo capitolo abbiamo introdotto gli specificatori d'accesso `public` e `private`. Prima di passare ai file di intestazione, dobbiamo considerare quali elementi del diagramma delle classi dovranno essere `public` e quali `private`.

Ingegneria del software 6.24

Per il principio del minimo privilegio, un elemento di una classe avrà visibilità di tipo `private`, a meno che non si dimostri la necessità che esso abbia visibilità di tipo `public`.

In questo capitolo abbiamo già sottolineato che i dati membro dovrebbero essere generalmente `private`, ma come trattare le funzioni? La risposta è che le funzioni membro corrispondono alle operazioni svolte da una classe e dunque possono essere invocate dai client della classe. Per questo motivo esse dovrebbero essere dichiarate come `public`. In UML la visibilità di tipo `public` si indica apponendo un segno "+" all'elemento in questione (sia esso funzione o dato membro), mentre la visibilità di tipo `private` si indica apponendo un segno "-"

La Figura 6.13 mostra il nostro diagramma delle classi aggiornato, con le notazioni relative alla visibilità. Come potete notare abbiamo aggiunto l'operazione `personArrives` alla classe `Floor` al nostro diagramma di sequenza di Figura 4.27.

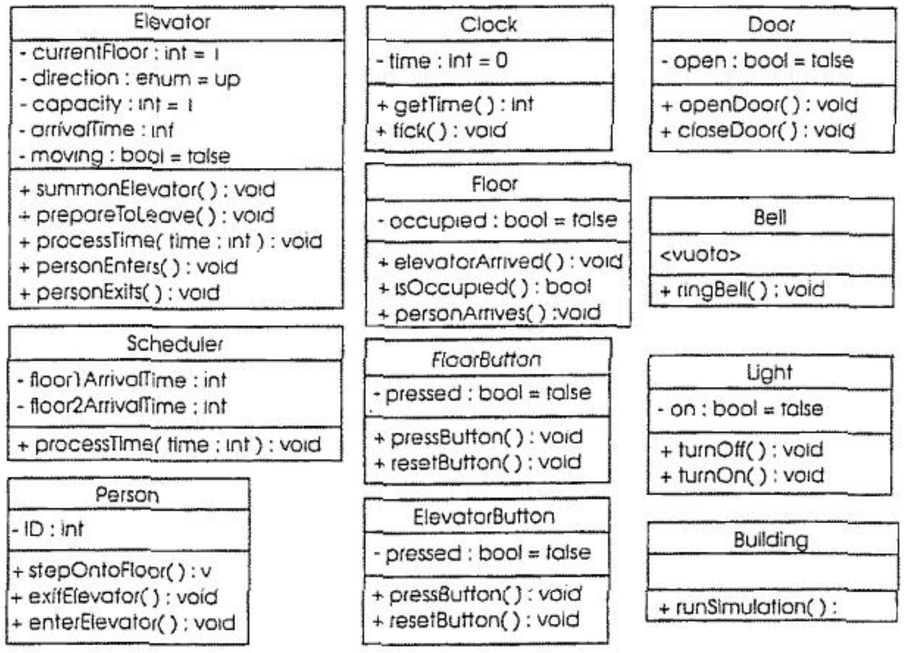


Figura 6.13 Diagramma delle classi completo con le indicazioni che riguardano la visibilità.

A questo punto, nello scrivere i file di intestazione possiamo automaticamente porre gli elementi contrassegnati con il segno "+" nelle sezioni `public` e quelli contrassegnati con il segno "-" nelle sezioni `private` delle dichiarazioni di ogni classe.

6.18.2 Implementazione: handle

Perché un oggetto della classe A possa comunicare con un oggetto della classe B, l'oggetto della classe A deve disporre di un *handle* (in inglese, *maniglia*) all'oggetto della classe B. Ciò significa che l'oggetto della classe A deve conoscere il nome dell'oggetto della classe B, oppure che l'oggetto della classe A deve disporre di un riferimento (Sezione 3.17) o di un puntatore (Capitolo 5) all'oggetto della classe B. La Figura 5.36 conteneva l'elenco delle collaborazioni tra gli oggetti del sistema: le classi nella colonna di sinistra hanno bisogno di un handle a ciascuna delle classi della colonna di destra, per poter inviar loro dei messaggi. La Figura 6.14 elenca gli handle di ciascuna classe sulla base delle informazioni che è possibile ricavare dalla Figura 5.36.

Nel corso di questo capitolo abbiamo illustrato come implementare gli handle in C++, come riferimenti o puntatori alle classi, e cogliamo l'occasione per ribadire che è preferibile utilizzare i riferimenti anziché i puntatori, dove è possibile. I riferimenti diventano poi attributi (dati) della classe. Finché non studieremo la composizione (Capitolo 7), non potremmo rappresentare tutti gli elementi della Figura 6.14. Discuteremo tra breve degli elementi problematici.

Classe	Handle
`Elevator`	`ElevatorButton, Bell, Floor, Door`
`Clock`	
`Scheduler`	`Person, Floor`
`Person`	`FloorButton, ElevatorButton, Elevator, Floor`
`Floor`	`FloorButton, Light`
`FloorButton`	`Elevator`
`ElevatorButton`	`Elevator`
`Door`	`Person`
`Bell`	
`Light`	
`Building`	`Clock, Scheduler, Elevator`

Figura 6.14 Elenco degli handle di ogni classe.

6.18.3 Implementazione: i file di intestazione delle classi

Ora che siamo a conoscenza di come programmare le classi in C++, siamo in grado di cominciare a scrivere il codice del simulatore di ascensore. In questa sezione ci concentreremo sui file di intestazione delle classi, mentre nella sezione "Pensare in termini di oggetti" del Capitolo 7 presenteremo il codice completo del simulatore. Infine nel Capitolo 9 modificheremo opportunamente il codice per inglobarvi il concetto di ereditarietà. Per evidenziare l'ordine di esecuzione di costruttori e distruttori, tutti i costruttori e i distruttori conterranno una linea di codice che visualizza un messaggio di esecuzione. I file di

intestazione contengono i prototipi dei costruttori e dei distruttori, mentre le rispettive implementazioni si trovano nei file `.cpp` che presenteremo nel capitolo seguente.

La Figura 6.15 mostra il file di intestazione della classe `Bell`. Facendo riferimento al diagramma delle classi (Figura 6.13), dichiariamo un costruttore, un distruttore (linee 8 e 9) e la funzione membro `ringBell` (linea 10), che hanno tutte visibilità di tipo `public`. Non abbiamo identificato altri elementi `public` o `private` di questa classe, per cui il suo file di intestazione è già completo.

```
1   // bell.h
2   // Definizione della classe Bell.
3   #ifndef BELL_H
4   #define BELL_H
5
6   class Bell {
7   public:
8   Bell(); // costruttore
9   ~Bell(); // distruttore
10  void ringBell(); // suona il campanello
11  };
12
13  #endif // BELL_H
```

Figura 6.15 File di intestazione della classe Bell.

La Figura 6.16 mostra il file di intestazione della classe `Clock`. Il file contiene un costruttore, un distruttore (linee 8 e 9) e le funzioni membro `tick()` e `getTime()` (linee 10 e 11), entrambe `public` (seguendo quanto prescritto in Figura 6.13). Implementiamo l'attributo `time` nel file di intestazione dichiarando il dato membro `time` di tipo `int` e visibilità `private` (linea 13). Durante la simulazione un oggetto della classe `Building` invocherà ad istanti regolari la funzione membro `getTime` di un oggetto della classe `Clock` per ottenere il valore corrente di `time`, e invocherà la funzione membro `tick` per incrementare `time`.

```
1   // clock.h
2   // Definizione della classe Clock.
3   #ifndef CLOCK_H
4   #define CLOCK_H
5
6   class Clock {
7   public:
8   Clock(); // costruttore
9   ~Clock(); // distruttore
10  void tick(); // passa all'istante successivo
11  int getTime(); // restituisce l'istante corrente del timer
12  private:
13  int time; // istante del timer
14  };
15
16  #endif // CLOCK_H
```

Figura 6.16 File di intestazione della classe Clock.

La Figura 6.17 mostra il file di intestazione della classe `Person`. Seguendo il diagramma delle classi in Figura 6.13 dichiariamo l'attributo `ID` (linea 16) e le operazioni `stepOntoFloor`, `enterElevator` ed `exitElevator` (linee 12–14). Dichiariamo poi la funzione membro `getID` con visibilità `public` (linea 10), che restituisce il numero identificativo della persona che sarà utilizzato per tener traccia delle persone durante la simulazione.

Gli oggetti della classe `Person` non sono creati all'inizio della simulazione, ma a caso e in modo dinamico, man mano che la simulazione procede, perciò dobbiamo implementarli diversamente dagli oggetti delle altre classi. Dopo aver discusso la creazione dinamica degli oggetti nel Capitolo 7, integreremo in modo significativo la dichiarazione della classe `Person`.

```
1   // person.h
2   // Definizione della classe Person
3   #ifndef PERSON_H
4   #define PERSON_H
5
6   class Person {
7   public:
8   Person( int ); // costruttore
9   ~Person(); // distruttore
10  int getID(); // restituisce l'ID della persona
11
12  void stepOntoFloor();
13  void enterElevator();
14  void exitElevator()
15  private:
16  int ID; // ID univoco della persona
17  };
18
19  #endif // PERSON_H
```

Figura 6.17 File di intestazione della classe Person.

La Figura 6.18 mostra il file di intestazione della classe `Door`. Dichiariamo un costruttore e un distruttore (linee 8 e 9), le funzioni membro `openDoor` e `closeDoor` di tipo `public` alle linee 11 e 12 ed infine il dato membro `open` di tipo `private` alla linea 14. La tabella in Figura 6.14 indica anche che la classe `Door` deve disporre di un handle alla classe `Person`, ma gli oggetti di questa classe sono creati dinamicamente, e per il nostro livello di conoscenza non sappiamo come implementare handle a oggetti dinamici. Rimandiamo dunque questo punto al Capitolo 7.

Il listato del file di intestazione della classe `Light` è riportato in Figura 6.19. Le informazioni che ricaviamo dal diagramma delle classi in Figura 6.13 ci suggeriscono di dichiarare le funzioni membro `turnOn` e `turnOff`, di visibilità `public`, e il dato membro `on` di tipo `bool`. In questo file di intestazione introduciamo anche qualcosa di nuovo alla nostra implementazione, cioè la necessità di distinguere tra oggetti diversi di una stessa classe. Abbiamo visto che nella simulazione ci sono due oggetti della classe `Light`, uno al primo piano e uno al secondo. Vogliamo distinguere questi due oggetti nell'output, per cui diamo loro un nome. Aggiungiamo quindi la linea 14

```
char *name; // piano della spia
```


alla sezione `private` della dichiarazione della classe. Aggiungiamo inoltre il parametro `char *` al costruttore (linea 8), in modo che possa inizializzare il nome di entrambi gli oggetti della classe `Light`.

```
1   // door.h
2   // Definizione della classe Door
3   #ifndef DOOR_H
4   #define DOOR_H
5
6   class Door {
7   public:
8   Door(); // costruttore
9   ~Door(); // distruttore
10
11  void openDoor();
12  void closeDoor();
13  private:
14  bool open; // aperta o chiusa
15  };
16
17  #endif // DOOR_H
```

Figura 6.18 File di intestazione della classe Door.

```
1   // light.h
2   // Definizione della classe Light.
3   #ifndef LIGHT_H
4   #define LIGHT_H
5
6   class Light {
7   public:
8   Light( char * ); // costruttore
9   ~Light(); // distruttore
10  void turnOn(); // accende la spia
11  void turnOff(); // spegne la spia
12  private:
13  bool on; // accesa o spenta
14  char *name;
15  };
16
17  #endif // LIGHT_H
```

Figura 6.19 File di intestazione della classe Light.

La Figura 6.20 mostra il file di intestazione della classe `Building`. La sezione `public` della classe contiene un costruttore, un distruttore e la funzione membro `runSimulation` secondo la Figura 6.13. Quando abbiamo identificato per la prima volta l'operazione `runSimulation` nel Capitolo 4, non sapevamo quale oggetto avrebbe invocato la funzione per iniziare la simulazione. Ora che sappiamo come utilizzare le classi in C++, sappiamo che un oggetto della classe `Building` sarà dichiarato nella funzione `main`, e da essa sarà invocata la funzione `runSimulation`. Scegliamo anche di includere un parametro di tipo

int nella dichiarazione di runSimulation, che specifica la durata in secondi della simulazione Il codice della sezione principale del programma sarà dunque:

```
Building building; // crea l'oggetto edificio
building.runSimulation(30); // invoca runSimulation per una durata di
30 secondi
```

La tabella in Figura 6.14 indica che la classe Building deve disporre di handle agli oggetti composti, ma non sappiamo ancora implementare handle di questo genere, perché non abbiamo ancora introdotto la composizione: dovremo rimandare l'argomento al prossimo capitolo (vedi commenti alle linee 14-18 in Figura 6.20).

```
1   // building.h
2   // Definizione della classe Building.
3   #ifndef BUILDING_H
4   #define BUILDING_H
5
6   class Building {
7   public:
8   Building(); // costruttore
9   ~Building(); // distruttore
10
11  // esegue la simulazione per la durata specificata
12  void runSimulation( int );
13  private:
14  // Nel Capitolo 7 mostreremo come includere:
15  // un oggetto della classe Clock
16  // un oggetto della classe Scheduler
17  // un oggetto della classe Elevator
18  // due oggetti della classe Floor
19  };
20
21  #endif // BUILDING_H
```

Figura 6.20 File di intestazione della classe Building.

La Figura 6.21 contiene il file di intestazione della classe ElevatorButton. Seguendo le indicazioni del diagramma delle classi in Figura 6.13, dichiariamo l'attributo pressed, le funzioni membro pressButton e resetButton, il costruttore e il distruttore. La Figura 6.14 indica che la classe ElevatorButton deve disporre di un handle alla classe Elevator, che è incluso alla linea 19

```
Elevator &elevatorRef;
```

come riferimento. Nel Capitolo 7 illustreremo come inviare messaggi all'ascensore tramite questo riferimento.

Un riferimento deve essere inizializzato, ma non possiamo assegnare un valore a un dato membro di una classe nel file di intestazione, perciò lo inizializzeremo nel costruttore: passeremo un riferimento ad Elevator come parametro del costruttore, alla linea 10.

La linea 6

```
class Elevator; // dichiarazione anticipata
```


è una *dichiarazione anticipata* della classe Elevator, che consente di dichiarare un riferimento a un oggetto della classe Elevator senza dover includere il file di intestazione della classe Elevator nel file di intestazione della classe ElevatorButton [Nota: utilizzando una dichiarazione anticipata (se possibile) anziché includere il file di intestazione completo contribuisce ad evitare un problema del preprocessore detto "inclusione circolare" Ne discuteremo in dettaglio nel Capitolo 7].

```
1   // elevatorButton.h
2   // Definizione della classe ElevatorButton.
3   #ifndef ELEVATORBUTTON_H
4   #define ELEVATORBUTTON_H
5
6   class Elevator; // dichiarazione anticipata
7
8   class ElevatorButton {
9   public:
10  ElevatorButton( Elevator & ); // costruttore
11  ~ElevatorButton(); // distruttore
12
13  void pressButton(); // preme il pulsante
14  void resetButton(); // rilascia il pulsante
15  private:
16  bool pressed; // stato del pulsante
17
18  // riferimento all'ascensore interno al pulsante
19  Elevator &elevatorRef;
20  };
21
22  #endif // ELEVATORBUTTON_H
```

Figura 6.21 File di intestazione della classe ElevatorButton.

La Figura 6.22 contiene il file di intestazione della classe FloorButton. Questo file di intestazione è identico a quello della classe ElevatorButton, tranne per il dato membro floorNumber, che è di tipo int e visibilità private. Per ragioni di output, gli oggetti della classe FloorButton devono sapere su che piano si trovano e, a tale scopo, il numero del piano viene passato come argomento del costruttore per l'inizializzazione (linea 10).

```
1   // floorButton.h
2   // Definizione della classe FloorButton.
3   #ifndef FLOORBUTTON_H
4   #define FLOORBUTTON_H
5
6   class Elevator; // dichiarazione anticipata
7
8   class FloorButton {
9   public:
10  FloorButton( int, Elevator & ); // costruttore
11  ~FloorButton(); // distruttore
12
```

Figura 6.22 File di intestazione della classe FloorButton (continua)

```
13    void pressButton(); // premi il pulsante
14    void resetButton(); // rilascia il pulsante
15    private:
16    int floorNumber; // piano del pulsante
17    bool pressed; // stato del pulsante
18
19    // riferimento all'ascensore interno al pulsante
20    Elevator &elevatorRef;
21    };
22
23    #endif // FLOORBUTTON_H
```

Figura 6.22 File di intestazione della classe FloorButton.

La Figura 6.23 mostra il file di intestazione della classe `Scheduler`. Alle linee 23 e 24

```
int floor1ArrivalTime;
int floor2ArrivalTime;
```

dichiariamo i dati membri `private` della classe `Scheduler` che corrispondono agli attributi che abbiamo individuato per questa classe (Figura 6.13). Alla linea 12 dichiariamo la funzione membro `processTime` di visibilità `public`, che corrisponde all'operazione che abbiamo identificato nella sezione "Pensare in termini di oggetti" del Capitolo 4.

```
1   // scheduler.h
2   // Definizione della classe Scheduler
3   #ifndef SCHEDULER_H
4   #define SCHEDULER_H
5
6   class Floor; // dichiarazione anticipata
7
8   class Scheduler {
9   public:
10  Scheduler( Floor &, Floor & ); // costruttore
11  ~Scheduler(); // distruttore
12  void processTime( int ); // imposta l'istante corrente
13  private:
14
15  // metodo che programma gli arrivi ai piani specificati
16  void scheduleTime( Floor & );
17
18  // metodo che ritarda gli arrivi ai piani specificati
19  void delayTime( Floor & );
20
21  Floor &floor1Ref;
22  Floor &floor2Ref;
23  int floor1ArrivalTime;
24  int floor2ArrivalTime;
25  };
26
27  #endif // SCHEDULER_H
```

Figura 6.23 File di intestazione della classe Scheduler.

Alle linee 15-19 dichiariamo le funzioni che abbiamo individuato nel diagramma di sequenza in Figura 4.27. Ognuna di esse prende come parametro un riferimento a un *oggetto della classe* `Floor`.

Notate che non abbiamo elencato queste funzioni sotto le operazioni (come funzioni membro `public`), perché questi metodi non sono invocati dagli oggetti client. Essi, infatti, sono utilizzati soltanto dalla classe `Scheduler` per effettuare operazioni interne, perciò è appropriato porli nella sezione `private` della dichiarazione.

Alle linee 21 e 22 dichiariamo gli handle individuati in Figura 6.14. Come abbiamo già detto, implementiamo ogni handle come riferimento a un oggetto della classe `Floor`. La classe `Scheduler` deve disporre di questi handle per poter inviare il messaggio `isOccupied` ai due piani durate la simulazione (cfr. diagramma in Figura 4.27). Dobbiamo prevedere anche una dichiarazione anticipata della classe `Floor` alla linea 6, per poterne dichiarare dei riferimenti.

La Figura 6.24 contiene i file di intestazione della classe `Floor`. Seguendo il diagramma in Figura 6.13, dichiariamo le funzioni membro `elevatorArrived`, `isOccupied` e `personArrives`, di tipo `public` e la funzione membro `elevatorLeaving` alla linea 26, anch'essa di tipo `public`. Abbiamo aggiunto questa funzione membro in modo tale che l'ascensore possa comunicare al piano quando si sta preparando per la corsa. L'ascensore invocherà l'operazione `elevatorLeaving` e il piano risponderà spegnendo la spia esterna.

```
1   // floor.h
2   // Definizione della classe Floor.
3   #ifndef FLOOR_H
4   #define FLOOR_H
5
6   class Elevator; // dichiarazione anticipata
7
8   class Floor {
9   public:
10  Floor( int, Elevator & ); // costruttore
11  ~Floor(); // distruttore
12
13  // restituisce true se il piano è occupato
14  bool isOccupied();
15
16  // restituisce il numero del piano
17  int getNumber();
18
19  // passa un handle alla nuova persona al piano
20  void personArrives();
21
22  // notifica al piano l'arrivo dell'ascensore
23  void elevatorArrived();
24
25  // notifica al piano la partenza dell'ascensore
26  void elevatorLeaving();
```

Figura 6.24 File di intestazione della classe Floor (continua)

```
27
28    // dichiarazione del componente di FloorButton (cfr  Capitolo 7)
29
30    private:
31    int floorNumber; // numero del piano
32    Elevator &elevatorRef; // handle all'ascensore
33    // dichiarazione del componente di Light (cfr  Capitolo 7)
34    };
35
36    #endif // FLOOR_H
```

Figura 6.24 File di intestazione della classe Floor.

Alla linea 31 aggiungiamo il dato membro `floorNumber`, di tipo `private`, che servirà per l'output, in modo analogo a `floorNumber` di `FloorButton`. Abbiamo anche previsto un parametro di tipo `int` al costruttore, perché inizializzi tale dato membro. Dichiariamo poi l'handle alla classe `Elevator` che abbiamo identificato nella Figura 6.14. Rimandiamo la dichiarazione dei membri che compongono la classe `Floor` (cfr. linee 28 e 33) al prossimo capitolo.

Il listato del file di intestazione della classe `Elevator` è presentato in Figura 6.25. Nella sezione `public` dichiariamo le operazioni `summonElevator`, `prepareToLeave` e `processTime` elencate in Figura 6.13. Per distinguere tra persone al piano e persone nell'ascensore rinominiamo le ultime due operazioni della classe `Elevator` chiamandole `passengerEnters` e `passengerExits` e le dichiariamo nella sezione `public` del file di intestazione. Dichiariamo infine un riferimento a ciascuno dei due piani (linee 38-39), ognuno dei quali sarà inizializzato dal costruttore (linea 10).

Nella sezione `private` del file di intestazione dichiariamo gli attributi `moving`, `direction`, `currentFloor` e `arrivalTime`, in accordo con la Figura 6.13. Non abbiamo bisogno di dichiarare l'attributo `capacity`, perché scriveremo il codice in modo tale che nell'ascensore possa trovarsi sempre al massimo una persona.

```
1    // elevator.h
2    // Definizione della classe Elevator.
3    #ifndef ELEVATOR_H
4    #define ELEVATOR_H
5
6    class Floor; // dichiarazione anticipata
7
8    class Elevator {
9    public:
10   Elevator( Floor &, Floor & ); // costruttore
11   ~Elevator(); // distruttore
12
13   // chiede all'ascensore il servizio a un piano particolare
14   void summonElevator( int );
15
```

Figura 6.25 File di intestazione della classe Elevator (continua)


```
16   // prepara l'ascensore alla corsa
17   void prepareToLeave();
18
19   // passa l'istante corrente all'ascensore
20   void processTime( int );
21
22   // notifica all'ascensore che sta salendo un passeggero
23   void passengerEnters();
```

Figura 6.25 File di intestazione della classe Elevator.

6.18.4 Conclusione

Nella sezione "Pensare in termini di oggetti" del prossimo capitolo presenteremo il codice C++ completo del simulatore. I concetti che illustreremo nel corso del capitolo ci serviranno a implementare le relazioni di composizione, la creazione dinamica di oggetti della classe `Person` e i dati membro e le funzioni `static` e `const`. Nel Capitolo 9 faremo riferimento al concetto di ereditarietà per migliorare ulteriormente la progettazione e l'implementazione del nostro simulatore.

Esercizi di autovalutazione

6.1 Completate le seguenti affermazioni:

a) La parola chiave _____ introduce la definizione di una struttura.
b) Ai membri di una classe si accede tramite l'operatore _____ e il nome di un oggetto della classe, oppure tramite l'operatore _____ e il puntatore a un oggetto della classe.
c) Ai membri di una classe specificati come _____ possono accedere soltanto le funzioni membro e le funzioni **`friend`** della classe.
d) Un _____ è una funzione membro speciale che serve a inizializzare i dati membro di una classe.
e) La modalità di accesso di default per i membri di una classe è _____
f) Una funzione _____ serve a modificare il valore di dati membro **`private`** di una classe.
g) _____ assegna un oggetto di una classe a un oggetto della stessa classe.
h) Le funzioni membro di una classe sono normalmente _____ mentre i dati membro sono normalmente _____
i) Una funzione _____ serve a leggere il valore di dati membro **`private`** di una classe.
j) L'insieme delle funzioni membro **`public`** di una classe costituisce la _____ della classe.
k) Si dice che l'implementazione di una classe è nascosta ai suoi client oppure che è _____
l) Le parole chiave _____ e _____ possono introdurre entrambe la definizione di una classe.
m) I membri di una classe specificati come _____ sono accessibili ovunque sia accessibile un oggetto della classe (ovvero nel raggio del suo scope).

6.2 Individuate l'errore/gli errori e spiegate come possono essere corretti.

a) Supponete di aver dichiarato il seguente prototipo nella classe `Time`.

```
void ~Time( int );
```

b) La seguente è una definizione parziale di **Time**.

```
class Time {
public:        // function prototypes
   private:
      int hour = 0;
   int minute = 0;
   int second = 0;
};
```

c) Supponete di aver dichiarato il seguente prototipo nella classe **Employee**.

```
int Employee( const char *, const char * );
```

Risposte agli esercizi di autovalutazione

6.1 a) **struct**. b) punto (**.**), freccia (**->**). c) **private**. d) costruttore. e) **private**. f) *set*. g) Copia membro a membro di default (eseguita dall'operatore di assegnamento). h) **public**, **private**. i) *get*. j) interfaccia. k) incapsulata. l) **class**, **struct**. m) **public**.

6.2 a) Errore: I distruttori non possono restituire un valore o ricevere degli argomenti.
Correzione: Cancellare il tipo restituito e il parametro **int** dalla dichiarazione.

b) Errore: I membri non possono essere inizializzati esplicitamente in una definizione di classe.
Correzione: Eliminare l'inizializzazione esplicita dalla definizione di classe e inizializzare i dati membro in un costruttore.

c) Errore: I costruttori non possono restituire un valore.
Correzione: Cancellare il tipo **int** restituito dal costruttore dalla dichiarazione.

Esercizi

6.3 A che serve l'operatore di risoluzione dello scope?

6.4 Mettete a confronto le nozioni di struttura e di classe in C++.

6.5 Scrivete un costruttore che sia in grado di utilizzare l'orario attuale prendendolo dalla funzione **time()**, dichiarata nel file di intestazione della libreria standard **time.h**. Il valore ottenuto dalla funzione servirà a inizializzare un oggetto della classe **Time**.

6.6 Definite la classe **Complex** per effettuare operazioni matematiche sui numeri complessi. Scrivete un programma di prova per verificare l'adeguatezza della vostra classe.
I numeri complessi hanno la seguente forma

```
Re + i*Im
```

dove Re è la *parte reale* del numero, *Im* è la cosiddetta *parte immaginaria* e *i* indica la radice quadrata di –1.
Rappresentate i dati **private** della classe tramite variabili a virgola mobile. Scrivete un costruttore che inizializzi ogni oggetto della classe quando è dichiarato. Il costruttore deve contenere dei valori di default, nel caso che non siano forniti inizializzatori. Ognuna delle quattro operazioni seguenti dovrebbe essere svolta da una funzione membro diversa:

a) Addizione di due numeri complessi: parti reali e parti immaginarie vanno sommate separatamente.
b) Sottrazione di due numeri complessi: la parte reale dell'operando destro è sottratta dalla parte reale dell'operando sinistra, e lo stesso vale per le parti immaginarie.
c) Visualizzazione di numeri complessi nella forma **(a, b)** dove **a** è la parte reale e **b** quella immaginaria.

6.7 *Definite la class* **Rational** *per effettuare operazioni matematiche sui numeri razionali, cioè i* numeri rappresentabili come quoziente di due numeri interi. Scrivete un programma di prova per verificare l'adeguatezza della vostra classe.
Rappresentate i dati **private** della classe con variabili intere: numeratore e denominatore. Il costruttore *deve inizializzare ogni oggetto della classe, quando viene dichiarato, e deve contenere dei valori di* default, nel caso non siano forniti inizializzatori. Il costruttore deve memorizzare i valori delle frazioni in forma ridotta: per es. la frazione 2/4 sarà memorizzata con 1 al numeratore e 2 al denominatore. Ognuna delle operazioni che segue dovrebbe essere svolta da una funzione membro diversa:

a) Addizione di due numeri **Rational**. Il risultato è memorizzato in forma ridotta.
b) Sottrazione di due numeri **Rational**. Il risultato è memorizzato in forma ridotta.
c) Moltiplicazione di due numeri **Rational**. Il risultato è memorizzato in forma ridotta.
d) Divisione di due numeri **Rational**. Il risultato è memorizzato in forma ridotta.
e) Visualizzazione dei numeri **Rational** nella forma **a/b**.
f) Visualizzazione dei numeri **Rational** nel formato a virgola mobile.

6.8 Modificate la classe **Time** del programma in Figura 6.10 includendo la funzione membro **tick** che incrementa l'orario di un oggetto **Time** di un'unità al secondo. L'oggetto **Time** deve sempre restare *in uno stato coerente. Scrivete un programma di prova che verifica il funzionamento della funzione* **tick** in un ciclo, visualizzando continuamente l'orario in formato standard. Verificate il corretto funzionamento per:

a) L'incremento del minuto successivo.
b) L'incremento dell'ora successiva.
c) L'incremento del giorno successivo (da 11:59:59 PM a 12:00:00 AM).

6.9 Modificate la classe **Date** in Figura 6.12 includendo un controllo degli errori per il valore degli inizializzatori dei dati membro **month**, **day** e **year**. Includete anche la funzione membro **nextDay** per *incrementare di uno il giorno. Un oggetto* **Date** *deve restare sempre in uno stato coerente. Scrivete un* programma di prova che verifica il funzionamento della funzione **nextDay** in un ciclo. Verificate il corretto funzionamento per:

a) L'incremento del mese successivo.
b) L'incremento dell'anno successivo.

6.10 Combinate la classe **Time** modificata dell'Esercizio 6.8 e la classe **Date** dell'Esercizio 6.9 in un'unica classe **DateAndTime** (nel Capitolo 9 parleremo dell'ereditarietà che ci aiuterà a compiere questa operazione velocemente e senza modificare le classi precedenti. Modificate la funzione **tick** in *modo che chiami* **nextDay** *se l'orario giunge al giorno successivo. Modificate le funzioni* **printStandard** e **printMilitary** in modo che visualizzino data e ora. Scrivete un programma di prova che verifica il funzionamento della nuova classe **DateAndTime**. Verificate l'orario al cambio di data

6.11 Modificate le funzioni *set* del programma in Figura 6.10 in modo che restituiscano dei valori di errore se si cerca di modificare un dato membro con un valore non valido.

6.12 Definite la classe **Rectangle**. Questa classe ha gli attributi **length** (lunghezza) e **width** (ampiezza), ognuno dei quali per default vale 1. Ha due funzioni membro che calcolano il perimetro e l'area del rettangolo e funzioni *set* e *get* per **length** e **width**. Le funzioni *set* devono verificare che **length** e **width** siano numeri a virgola mobile compresi tra 0.0 e 20.0.

6.13 Definite una versione più sofisticata della classe **Rectangle** dell'Esercizio 6.12. Questa classe contiene le coordinate cartesiane dei quattro angoli del rettangolo. Il costruttore chiama una funzione *set che accetta quattro coppie di coordinate e verifica che ognuna di esse si trovi nel primo quadrante* (cioè verifichi che x e y siano maggiori di zero) e che tutti i valori delle coppie **x** e **y** non siano maggiori di 20.0. La funzione *set* verifica anche se le coordinate specificate individuano effettivamente un

rettangolo. Altre funzioni membro calcolano la lunghezza, l'ampiezza, il perimetro e l'area. La lunghezza è la più grande delle due dimensioni. Includete la funzione predicativa **square** che determina se il rettangolo è un quadrato.

6.14 Modificate la classe **Rectangle** dell'Esercizio 6.13 includendo la funzione **draw** che visualizza il rettangolo in un'area 25x25 nel primo quadrante, in cui si trova il rettangolo. Includete la funzione **setFillCharacter** per specificare il carattere con cui riempire il rettangolo. Includete la funzione **setPerimeterCharacter** per specificare il carattere con cui disegnare il bordo del rettangolo. Un ulteriore esercizio potrebbe essere quello di includere altre funzioni membro per ridimensionare in scala il rettangolo, per ruotarlo e spostarlo in modo che il proprio centro si trovi in una posizione specificav passata come parametro.

6.15 Definite la classe **HugeInteger** che usa un array di 40 elementi numerici per memorizzare numeri interi composti di un numero massimo di 40 cifre ciascuno. Scrivete le funzioni membro **inputHugeInteger**, **outputHugeInteger**, **addHugeIntegers** (somma) e **substractHugeIntegers** (sottrazione). Per il confronto di due oggetti **HugeInteger** scrivete le funzioni **isEqualTo** (è uguale a), **isNotEqualTo** (non è uguale a), **isGreaterThan** (è maggiore di), **isLessThan** (è minore di), **IsGreaterThanOrEqualTo** (è maggiore o uguale a) e **isLessThanOrEqualTo** (è minore o uguale a), ognuna delle quali è una funzione predicativa che restituisce **true** se la relazione che esprime è vera. Scrivete la funzione predicativa **isZero** (è nullo). Inoltre, potreste scrivete anche le funzioni **multiplyHugeIntegers** (moltiplicazione), **divideHugeIntegers** (divisione) e **modulusHugeIntegers** (modulo).

6.16 Definite la classe **TicTacToe** che vi consentirà di scrivere un programma completo per giocare a Tris. La classe contiene un array di interi 3x3 **private**. Il costruttore deve azzerare tutte le caselle dello schema iniziale. Il gioco si svolge tra due giocatori. Le mosse del primo giocatore sono contrassegnate da 1, quelle del secondo da 2. Ogni giocatore può muovere soltanto su una casella vuota. Dopo ogni mossa determinate se c'è un vincitore o se si deve continuare. Inoltre, potreste modificare il programma in modo che sia il computer a scegliere le mosse di un giocatore consentendo al giocatore umano di indicare se vuole muovere per primo o meno. Infine, se proprio volete strafare, provate a scrivere un programma che gioca il Tris su uno schema tridimensionale 4x4x4 (Attenzione: questo progetto è estremamente complicate e potrebbe impegnarvi per intere settimane!).

CAPITOLO 7

Le classi: seconda parte

Obiettivi

- Imparare a gestire dinamicamente gli oggetti
- Comprendere l'uso di const con gli oggetti e le funzioni membro
- Comprendere le funzioni e le classi friend
- Imparare a utilizzare i dati e le funzioni membro static
- Imparare ad utilizzare il puntatore this
- Comprendere il concetto di classe container
- Comprendere il concetto di classe iteratore che consente l'indicizzazione degli elementi di una classe container

7.1 Introduzione

In questo capitolo continueremo a esplorare le caratteristiche delle classi e dell'astrazione dei dati. Parleremo anche di argomenti avanzati e getteremo le fondamenta per la discussione dell'overloading degli operatori, oggetto del prossimo Capitolo. La trattazione che abbiamo cominciato nello scorso capitolo vi incoraggerà a utilizzare gli oggetti, in quella che chiamiamo la *programmazione basata sugli oggetti (OBP)*. Nei Capitoli 9 e 10 introdurremo i concetti di ereditarietà e polimorfismo, grazie ai quali sarete in grado di utilizzare appieno le tecniche di *programmazione orientata agli oggetti (OOP)*. In questo e nei prossimi capitoli utilizzeremo il più possibile le stringhe in stile C che abbiamo introdotto nel Capitolo 5. Speriamo così di farvi acquisire una certa familiarità con l'argomento, peraltro complesso, dei puntatori; ciò vi preparerà alla vostra vita professionale in cui dovrete fare i conti con l'enorme bagaglio di codice C ancora in uso. Nel Capitolo 8 del volume Tecniche Avanzate parleremo delle nuove stringhe disponibili in C++, concepite a tutti gli effetti come oggetti di una classe. In questo modo saprete trattare un oggetto stringa in entrambi i modi.

7.2 Gli oggetti e le funzioni membro costanti

Abbiamo parlato diverse volte del principio del minimo privilegio, che è uno dei concetti fondamentali dell'ingegneria del software. In questa sezione vediamo come può essere applicato agli oggetti.

Alcuni oggetti devono poter essere modificati mentre altri no. Il programmatore può utilizzare la parola chiave **const** per indicare che un oggetto non deve essere modificato, e che qualsiasi tentativo di cambiarlo deve essere segnalato come errore di sintassi.

Per esempio,

```
const Time noon( 12, 0, 0 );
```

dichiara l'oggetto costante `noon` della classe `Time` e lo inizializza a `12` (mezzogiorno).

Ingegneria del software 7.1

La dichiarazione di un oggetto costante è un'applicazione del principio del minimo privilegio: tutti i tentativi di modificare l'oggetto sono intercettati in fase di compilazione, e quindi non potranno causare degli errori in fase di esecuzione.

Ingegneria del software 7.2

L'utilizzo del costrutto `const` costituisce un argomento importante nella progettazione delle classi e nella scrittura di programmi.

Obiettivo efficienza 7.1

Se dichiarate variabili e oggetti costanti non avrete soltanto programmi di migliore qualità dal punto di vista dell'ingegneria del software, ma anche dal punto dell'efficienza: i moderni compilatori, infatti, sono in grado di ottimizzare il codice che effettua operazioni su dati costanti, cosa che non possono fare nel caso delle comuni variabili.

Il compilatore non consente chiamate a funzioni membro per gli oggetti costanti, a meno che esse non siano dichiarate come costanti. Le funzioni membro costanti non possono effettuare modifiche sull'oggetto.

Una funzione si dichiara come costante *sia* nel prototipo *sia* nella definizione, inserendo la parola chiave `const` al termine della lista di parametri, e nella definizione prima della parentesi graffa aperta che indica l'inizio del corpo della funzione. Per esempio la seguente funzione membro della classe `A`

```
int A::getValue() const { return privateDataMember };
```

restituisce semplicemente il valore di un dato membro dell'oggetto, ed è dichiarata giustamente come `const`.

Errore tipico 7.1

Se una funzione membro costante modifica un dato membro di un oggetto, il compilatore segnalerà un errore di sintassi.

Errore tipico 7.2

Se una funzione membro costante chiama una funzione membro non costante nella stessa istanza della classe, il compilatore segnalerà un errore di sintassi.

Errore tipico 7.3

Se invocate una funzione membro non costante su un oggetto costante commettete un errore di sintassi.

Ingegneria del software 7.3

Una funzione membro costante può subire overloading da parte di una versione non costante della funzione. Il compilatore deciderà volta per volta quale delle due funzioni chiamare, a seconda se l'oggetto sarà costante o meno.

A questo punto si può porre un problema riguardante l'uso di costruttori e distruttori con oggetti `const`, poiché in molti casi essi devono modificare in qualche modo gli oggetti. Non bisogna dichiarare come `const` i costruttori e i distruttori di oggetti costanti. Un costruttore deve poter modificare l'oggetto, in modo che i dati contenuti in esso siano inizializzati correttamente e lo stesso vale per i distruttori, i quali hanno bisogno di effettuare le operazioni finali prima che un oggetto sia distrutto e che la memoria occupata sia restituita al sistema.

Errore tipico 7.4

Se dichiarate un costruttore o un distruttore come `const` commettete un errore di sintassi.

Il programma in Figura 7.1 istanzia due oggetti di tipo `Time`, uno costante e uno non costante. Il programma tenta di modificare l'oggetto costante `noon` con le funzioni membro non costanti `setHour` (linea 100) e `printStandard` (linea 106). Abbiamo incluso nel programma anche le altre tre combinazioni possibili nell'uso di `const` tra funzioni e oggetti: una funzione membro non costante su un oggetto non costante (linea 98), una funzione membro costante su un oggetto non costante (linea 102) e infine una funzione membro costante su un oggetto costante (linee 104 e 105). La finestra di output mostra i messaggi generati da un compilatore per la chiamata di funzioni membro non costanti su oggetti costanti.

Buona abitudine 7.1

Dichiarate `const` tutte le funzioni membro che non effettuano modifiche sull'oggetto, in modo che possiate utilizzarle anche sugli oggetti costanti, se è necessario.

```
1  // Fig. 7.1: time5.h
2  // Dichiarazione della classe Time.
3  // Funzioni membro definite in time5.cpp
4  #ifndef TIME5_H
5  #define TIME5_H
6
7  class Time {
8  public:
9     Time( int = 0, int = 0, int = 0 );  // costruttore di default
10
11 // funzioni set
12    void setTime( int, int, int );  // imposta time
13    void setHour( int );            // imposta hour
14    void setMinute( int );          // imposta minute
15    void setSecond( int );          // imposta second
16
17    // funzioni get (di norma dichiarate come const)
18    int getHour() const;     // restituisce hour
19    int getMinute() const;   // restituisce minute
20    int getSecond() const;   // restituisce second
21
```

Figura 7.1 La classe Time con oggetti e funzioni membro const (continua)

```
22    // funzioni print (di norma dichiarate come const)
23    void printMilitary() const;  // visualizz. in formato militare
24    void printStandard();        // visualizz. in formato standard
25 private:
26    int hour;             // 0 - 23
27    int minute;           // 0 - 59
28    int second;           // 0 - 59
29 };
30
31 #endif

32 // Fig. 7.1: time5.cpp
33 // Definizioni delle funzioni membro per la classe Time.
34 #include <iostream.h>
35 #include "time5.h"
36
37 // Costruttore per inizializzare i dati private.
38 // I valori di default sono 0 (cfr. la definizione di classe).
39 Time::Time( int hr, int min, int sec )
40    { setTime( hr, min, sec ); }
41
42 // Impostazione dei valori di hour, minute e second.
43 void Time::setTime( int h, int m, int s )
44 {
45    setHour( h );
46    setMinute( m );
47    setSecond( s );
48 }
49
50 // Imposta hour
51 void Time::setHour( int h )
52    { hour = ( h >= 0 && h < 24 ) ? h : 0; }
53
54 // Imposta minute
55 void Time::setMinute( int m )
56    { minute = ( m >= 0 && m < 60 ) ? m : 0; }
57
58 // Imposta second
59 void Time::setSecond( int s )
60    { second = ( s >= 0 && s < 60 ) ? s : 0; }
61
62 // Restituisce hour
63 int Time::getHour() const { return hour; }
64
65 // Restituisce minute
66 int Time::getMinute() const { return minute; }
```

Figura 7.1 La classe Time con oggetti e funzioni membro const (continua)


```
67
68 // Restituisce second
69 int Time::getSecond() const { return second; }
70
71 // Visualizza l'orario in formato militare: HH:MM:SS
72 void Time::printMilitary() const
73 {
74    cout << ( hour < 10 ? "0" : "" ) << hour << ":"
75         << ( minute < 10 ? "0" : "" ) << minute;
76 }
77
78 // Visualizza l'orario in formato standard: HH:MM:SS AM (o PM)
79 void Time::printStandard()
80 {
81    cout << ( ( hour == 12 ) ? 12 : hour % 12 ) << ":"
82         << ( minute < 10 ? "0" : "" ) << minute << ":"
83         << ( second < 10 ? "0" : "" ) << second
84         << ( hour < 12 ? " AM" : " PM" );
85 }

86 // Fig. 7.1: fig07_01.cpp
87 // Tentativo di accedere a un oggetto const
88 // con funzioni membro non const.
89 #include <iostream.h>
90 #include "time5.h"
91
92 int main()
93 {
94    Time wakeUp( 6, 45, 0 );       // oggetto non costante
95    const Time noon( 12, 0, 0 );   // oggetto costante
96
97                                   // FUNZIONE MEMBRO      OGGETTO
98    wakeUp.setHour( 18 );          // non const            non const
99
100   noon.setHour( 12 );            // non const            const
101
102   wakeUp.getHour();              // const                non const
103
104   noon.getMinute();              // const                const
105   noon.printMilitary();          // const                const
106   noon.printStandard();          // non const            const
107   return 0;
108 }
```

Figura 7.1 La classe Time con oggetti e funzioni membro const (continua)

```
Compiling Fig07_01.cpp
Fig07_01.cpp(15) : error : 'setHour'
    cannot convert 'this' pointer from
    'const class Time' to 'class Time &'
    Conversion loses qualifiers
Fig07_01.cpp(21) : error : 'printStandard'
    cannot convert 'this' pointer from
    'const class Time' to 'class Time &'
    Conversion loses qualifiers
```

Figura 7.1 La classe Time con oggetti e funzioni membro const.

Anche se un costruttore deve essere una funzione membro non costante, può essere chiamato anche per un oggetto costante. La definizione del costruttore di `Time` nelle linee 39 e 40

```
Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }
```

mostra che il costruttore di `Time` chiama un'altra funzione membro non costante (`setTime`) per inizializzare l'oggetto `Time`. È consentito, infatti, chiamare una funzione membro non costante dal costruttore di un oggetto costante.

Ingegneria del software 7.4

Un oggetto `const` *non può essere modificato tramite un assegnamento, per cui deve necessariamente essere inizializzato. Se un dato membro di una classe è dichiarato come* `const`*, occorre un inizializzatore per fornire al costruttore il valore iniziale di tale membro.*

Osservate che la linea 106 (linea 21 nel file sorgente Fig07_01.cpp)

```
noon.printStandard();  // non const          const
```

genera un errore del compilatore anche se la funzione membro `printStandard` di `Time` non modifica l'oggetto per cui viene invocata.

La Figura 7.2 illustra l'uso dell'inizializzatore per il dato membro costante `increment` della classe `Increment`. Il costruttore di `Increment` è modificato in questo modo:

```
Increment::Increment( int c, int i )
   : increment( i )
{ count = c; }
```

La notazione `: increment( i )` inizializza `increment` con il valore `i`. Se sono necessari più inizializzatori, basta includerli in una lista di valori separati da virgole subito dopo il segno di due punti. Questa sintassi può servire a inizializzare anche tutti gli altri dati membro, ma i dati `const` e i riferimenti *devono* necessariamente essere inizializzati in questo modo. In seguito vedremo che gli oggetti membro devono essere inizializzati con questa sintassi.

Nel Capitolo 9, che introduce l'ereditarietà, vedremo che la sintassi degli inizializzatori di membro è seguita anche dalle porzioni di classe base di una classe derivata.

Collaudo e messa a punto 7.1

Dichiarate sempre `const` *le funzioni membro che non modificano l'oggetto. Così facendo delegherete il controllo di un buon numero di errori al compilatore.*

```
1   // Fig. 7.2: fig07_02.cpp
2   // Utilizzo di un inizializzatore di membro per inizializzare
3   // un dato costante di un tipo predefinito.
4
5   #include <iostream.h>
6
7   class Increment {
8   public:
9      Increment( int c = 0, int i = 1 );
10     void addIncrement() { count += increment; }
11     void print() const;
12
13  private:
14     int count;
15     const int increment;          // dato membro const
16  };
17
18  // Costruttore della classe Increment
19  Increment::Increment( int c, int i )
20     : increment( i )   // inizializzatore del membro const
21  { count = c; }
22  // Visualizza i dati
23  void Increment::print() const
24  {
25     cout << "count = " << count
26          << ", increment = " << increment << endl;
27  }
28
29  int main()
30  {
31     Increment value( 10, 5 );
32
33     cout << "Before incrementing: ";
34     value.print();
35
36     for ( int j = 0; j < 3; j++ ) {
37        value.addIncrement();
38        cout << "After increment " << j << ": ";
39        value.print();
40     }
41
42     return 0;
43  }
```

Figura 7.2 Uso di un inizializzatore di un membro costante di un tipo di dato predefinito (continua)

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Figura 7.2 Uso di un inizializzatore di un membro costante di un tipo di dato predefinito.

La Figura 7.3 illustra gli errori segnalati da un compilatore quando il programma tenta di inizializzare `increment` in un assegnamento anziché con un inizializzatore.

```
1  // Fig. 7.3: fig07_03.cpp
2  // Tentativo di inizializzare un dato costante di un tipo
3  // predefinito tramite un assegnamento.
4  #include <iostream.h>
5
6  class Increment {
7  public:
8     Increment( int c = 0, int i = 1 );
9     void addIncrement() { count += increment; }
10    void print() const;
11 private:
12    int count;
13    const int increment;
14 };
15
16 // Costruttore della classe Increment
17 Increment::Increment( int c, int i )
18 {           // Il membro costante 'increment' non è inizializzato
19    count = c;
20    increment = i;  // ERRORE: Non è possibile modificare un
21                    // oggetto costante
22 }
23
24 // Visualizza i dati
25 void Increment::print() const
26 {
27    cout << "count = " << count
28         << ", increment = " << increment << endl;
29 }
30
31 int main()
32 {
32    Increment value( 10, 5 );
33
34    cout << "Before incrementing: ";
35    value.print();
36
```

Figura 7.3 Tentativo illecito di inizializzare un dato membro costante di un tipo predefinito tramite un'istruzione di assegnamento (continua)


```
37    for ( int j = 0; j < 3; j++ ) {
38       value.addIncrement();
39       cout << "After increment " << j << ": ";
40       value.print();
41    }
42
43    return 0;
44 }
```

```
Compiling...
Fig7_3.cpp
Fig7_3.cpp(18) : error: 'increment' :
    must be initialized in constructor base/member
    initializer list
Fig7_3.cpp(20) : error: l-value specifies const object
```

Figura 7.3 Tentativo illecito di inizializzare un dato membro costante di un tipo predefinito tramite un'istruzione di assegnamento.

Errore tipico 7.5

Se omettete l'inizializzatore di un membro `const` *commettete un errore di sintassi.*

Ingegneria del software 7.5

I membri costanti di una classe (oggetti e variabili dichiarati come `const`*) devono essere inizializzati con degli inizializzatori appositi seguendo la sintassi sopraccitata: gli assegnamenti non sono consentiti.*

Osservate che la funzione `print` alla linea 25 è dichiarata come `const`. È una cosa ragionevole, anche se può sembrare strano, perché probabilmente non avremo mai a che fare con un oggetto `Increment` costante.

Ingegneria del software 7.6

È una buona abitudine dichiarare tutte le funzioni membro di una classe che non modificano l'oggetto su cui operano come `const`*. Questa può sembrare una stranezza se non avete alcuna intenzione di creare oggetti* `const` *di tale classe; tuttavia avrete un beneficio dal punto di vista del debugging: infatti se modificate accidentalmente un oggetto in tali funzioni, il compilatore ve lo segnalerà con un errore di sintassi.*

Collaudo e messa a punto 7.2

I linguaggi come il C++ evolvono continuamente e, probabilmente, le nuove versioni del linguaggio conterranno nuove parole chiave. Per questo motivo evitate di utilizzare come identificatori parole che hanno probabilità di diventarlo: pensate ad esempio alla parola "object" Oggi "object" non è una parola chiave, ma potrebbe facilmente diventarlo. Se questa nostra previsione si realizzasse, tutti i programmi che contengono variabili di nome "object" non potrebbero più essere compilati correttamente.

Il C++ ha una nuova parola chiave, `mutable`, che influenza il trattamento degli oggetti `const` in un programma. Torneremo su `mutable` nel Capitolo 10 del volume Tecniche Avanzate.

7.3 Il concetto di composizione: oggetti che diventano membri di altre classi

Una classe `AlarmClock` (in inglese `sveglia`) deve sapere quando far suonare la sveglia, perciò non sarebbe una cattiva idea includere un oggetto `Time` come membro di un oggetto `AlarmClock`. Così facendo sfruttiamo una funzionalità del C++ detta *composizione*: la possibilità per una classe di contenere come membri oggetti di altre classi.

Ingegneria del software 7.7

La funzionalità della composizione permette di scrivere software riutilizzabile, perché gli oggetti di altre classi possono essere riutilizzati nella creazione di nuove classi.

Quando si crea un oggetto viene chiamato automaticamente il suo costruttore, per cui occorre specificare come devono essere passati gli argomenti ai costruttori degli oggetti membro. Gli oggetti membro sono creati nell'ordine in cui sono dichiarati e non nell'ordine in cui sono elencati nella lista di inizializzatori del costruttore, e comunque prima degli oggetti che li contengono. Quest'ultimi sono detti anche oggetti *host*.

La Figura 7.4 mostra la composizione sulle classi `Employee` *e* `Date`. *La classe* `Employee` contiene i dati membro privati `firstName` (nome), `lastName` (cognome), `birthDate` (data di nascita) e `hireDate` (data di assunzione). I membri `birthDate` e `hireDate` sono oggetti `const` della classe `Date`, che contiene i dati membro privati `month` (mese), `day` (giorno) e `year` (anno). Il programma istanzia un oggetto `Employee`, inizializza i suoi dati membro e infine li invia in output. Osservate la sintassi dell'intestazione del costruttore di `Employee`:

```
Employee::Employee( char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
   : birthDate( bmonth, bday, byear ),
     hireDate( hmonth, hday, hyear )
```

Il costruttore riceve otto argomenti (`fname`, `lname`, `bmonth`, `bday`, `byear`, `hmonth`, `hday` e `hyear`). Il segno di due punti (`:`) presente nell'intestazione separa gli inizializzatori di membro dalla lista dei parametri. Gli inizializzatori di membro indicano gli argomenti di `Employee` da passare ai costruttori degli oggetti membro. Gli argomenti `bmonth`, `bday` e `byear` sono passati al costruttore di `birthDate`, mentre gli argomenti `hmonth`, `hday` e `hyear` sono passati al costruttore di `hireDate`. Le virgole separano i diversi inizializzatori di membro.

```
1  // Fig. 7.4: date1.h
2  // Dichiarazione della classe Date.
3  // Funzioni membro definite in date1.cpp
4  #ifndef DATE1_H
5  #define DATE1_H
6
7  class Date {
8  public:
9     Date( int=1, int = 1, int = 1900 ); // costruttore di default
```

Figura 7.4 Uso degli inizializzatori per gli oggetti membro (continua)


```
10    void print() const;  // visualizza la data nel formato m/g/a
11    ~Date();  // fornito per confermare l'ordine di distruzione
12 private:
13    int month;  // 1-12
14    int day;    // 1-31 dipende del mese
15    int year;   // qualsiasi anno
16    // funzione di utilità per verificare la correttezza di day
17    // sulla base di month e year
18    int checkDay( int );
19 };
20
21 #endif

22 // Fig. 7.4: date.cpp
23 // Definizioni delle funzioni membro della classe Date.
24 #include <iostream.h>
25 #include "date1.h"
26
27 // Costruttore: conferma un valore corretto per month;
28 // chiama la funzione di utilità checkDay per confermare un
29 // valore corretto di day.
30 Date::Date( int mn, int dy, int yr )
31 {
32    if ( mn > 0 && mn <= 12 )     // verifica la validità di month
33       month = mn;
34    else {
35       month = 1;
36       cout << "Month " << mn << " invalid. Set to month 1.\n";
37    }
38
39    year = yr;              // dovrebbe verificare la validità di yr
40    day = checkDay( dy ); // verifica la validità di day
41
42    cout << "Date object constructor for date ";
43    print();   // interessante: una funzione print senza argomenti
44    cout << endl;
45 }
46
47 // Visualizza l'oggetto Date nella forma month/day/year
48 void Date::print() const
49    { cout << month << '/' << day << '/' << year; }
50
51 // Distruttore: fornito per confermare l'ordine di distruzione
52 Date::~Date()
53 {
54    cout << "Date object destructor for date ";
55    print();
56    cout << endl;
57 }
```

Figura 7.4 Uso degli inizializzatori per gli oggetti membro (continua)

```
58
59 // Funzione di utilità che conferma un valore corretto per day
60 // che dipende da month e year.
61 // L'anno 2000 è bisestile?
62 int Date::checkDay( int testDay )
63 {
64    static const int daysPerMonth[ 13 ] =
65       {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
66
67    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
68       return testDay;
69
70    if ( month == 2 &&  //Febbraio:Controlla se l'anno è bisestile
71         testDay == 29 &&
72         ( year % 400 == 0 ||                          // anno 2000?
73          ( year % 4 == 0 && year % 100 != 0 ) ) ) //anno 2000?
74       return testDay;
75
76    cout << "Day " << testDay << " invalid. Set to day 1.\n";
77
78    return 1;  // lascia l'oggetto in uno stato coerente in caso
79               //  di valore scorretto
80 }
81 // Fig. 7.4: emply1.h
82 // Dichiarazione della classe Employee.
83 // Funzioni membro definite in emply1.cpp
84 #ifndef EMPLY1_H
85 #define EMPLY1_H
86
87 #include "date1.h"
88
89 class Employee {
90 public:
91    Employee( char *, char *, int, int, int, int, int, int );
92    void print() const;
93    ~Employee();// fornito per confermare l'ordine di  distruzione
94 private:
95    char firstName[ 25 ];
96    char lastName[ 25 ];
97    const Date birthDate;
98    const Date hireDate;
99 };
100
101 #endif
```

Figura 7.4 Uso degli inizializzatori per gli oggetti membro (continua)


```
102 // Fig. 7.4: emply1.cpp
103 // Definizioni delle funzioni membro della classe Employee.
104 #include <iostream.h>
105 #include <string.h>
106 #include "emply1.h"
107 #include "date1.h"
108
109 Employee::Employee( char *fname, char *lname,
110                     int bmonth, int bday, int byear,
111                     int hmonth, int hday, int hyear )
112    : birthDate( bmonth, bday, byear ),
113      hireDate( hmonth, hday, hyear )
114 {
115    // copia fname in firstName e si assicura che c'entri
116    int length = strlen( fname );
117    length = ( length < 25 ? length : 24 );
118    strncpy( firstName, fname, length );
119    firstName[ length ] = '\0';
120
121    // copia lname in lastName e si assicura che c'entri
122    length = strlen( lname );
123    length = ( length < 25 ? length : 24 );
124    strncpy( lastName, lname, length );
125    lastName[ length ] = '\0';
126
127    cout << "Employee object constructor: "
128         << firstName << ' ' << lastName << endl;
129 }
130 void Employee::print() const
131 {
132    cout << lastName << ", " << firstName << "\nHired: ";
133    hireDate.print();
134    cout << "  Birth date: ";
135    birthDate.print();
136    cout << endl;
137 }
138
139 // Distruttore: fornito per confermare l'ordine di distruzione
140 Employee::~Employee()
141 {
142    cout << "Employee object destructor: "
143         << lastName << ", " << firstName << endl;
144 }
145 // Fig. 7.4: fig07_04.cpp - Mostra l'uso della composizione:
146 // un oggetto che ha oggetti come membri.
147 #include <iostream.h>
148 #include "emply1.h"
149
```

Figura 7.4 Uso degli inizializzatori per gli oggetti membro (continua)

```
150  int main()
151  {
152     Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
153
154     cout << '\n';
155     e.print();
156
157     cout << "\nTest Date constructor with invalid values:\n";
158     Date d( 14, 35, 1994 );  // valori di Date non validi
159     cout << endl;
160     return 0;
161  }
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Jones
Jones, Bob
Hired: 3/12/1988  Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994
Date object destructor for date 1/1/1994
Employee object destructor: Jones, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

Figura 7.4 Uso degli inizializzatori per gli oggetti membro

Ricordate che la lista degli inizializzatori di membro contiene anche i valori dei membri `const` e dei riferimenti. Nel Capitolo 9 vedremo che sono inizializzate così anche le porzioni della classe base presenti negli oggetti di classi derivate. Le classi `Date` e `Employee` includono ognuna un distruttore che visualizza un messaggio quando viene distrutto un oggetto della rispettiva classe. Questo ci consente di avere una conferma nell'output del fatto che gli oggetti sono costruiti dall'interno verso l'esterno e distrutti in ordine inverso: nel nostro caso gli oggetti membro `Date` saranno distrutti dopo l'oggetto `Employee` che li contiene.

Un oggetto membro non deve necessariamente essere inizializzato in modo esplicito con un inizializzatore di membro; in realtà lo si può omettere e in tal caso verrà chiamato implicitamente il costruttore di default dell'oggetto membro. I valori impostati dal costruttore di default (se sono presenti) possono essere modificati per mezzo delle funzioni *set*.

Errore tipico 7.6

Se non scrivete un inizializzatore per un oggetto membro e nella classe di tale oggetto membro non esiste un costruttore di default, commettete un errore di sintassi.

Obiettivo efficienza 7.2

Inizializzate gli oggetti membro in modo esplicito tramite inizializzatori di membro. Così facendo, eviterete lo spreco di tempo che comporterebbe la "doppia inizializzazione" degli oggetti membro: la prima del costruttore di default e la seconda delle funzioni set *che inizializzano l'oggetto membro.*

Ingegneria del software 7.8

Se una classe contiene un oggetto di un'altra classe e tale oggetto membro viene reso `public`*, l'incapsulamento e l'occultamento dei dati membro* `private` *dell'oggetto non risultano violati. L'unica conseguenza è che sarà possibile accedere indirettamente ai membri pubblici dell'oggetto contenuto.*

Osservate la chiamata a `print`, funzione membro di `Date`, alla linea 43. Molte funzioni membro non hanno bisogno di alcun argomento: il motivo è che ogni funzione membro contiene un *handle* implicito, nella forma di puntatore, all'oggetto su cui deve operare. Nella Sezione 7.5 parleremo di questo puntatore che si chiama `this`.

Nella prima versione della classe `Employee`, per semplicità, abbiamo utilizzato due array di 25 caratteri per rappresentare il nome e il cognome di un `Employee`. Se i nomi sono molto più brevi di 24 caratteri si ha un notevole spreco di spazio (ricordate che l'ultimo carattere è sempre destinato al terminatore nullo `'\0'`). Inoltre non è possibile memorizzare nomi più lunghi di 24 caratteri, perché subiranno un troncamento per poter essere contenuti nell'array. In seguito presenteremo una nuova versione di `Employee` in grado di creare in modo dinamico l'esatta quantità di spazio per contenere il nome e il cognome. Potremmo utilizzare anche due oggetti di tipo `string` per memorizzare i nomi: presenteremo in dettaglio la classe `string` della libreria standard nel Capitolo 8 del volume Tecniche Avanzate.

7.4 Le funzioni e le classi friend

Una *funzione* `friend` (in inglese "amica") di una classe è definita al di fuori dello scope di tale classe, ma ha diritto ad accedere ai membri `private` di essa (e come vedremo in seguito anche ai dati membro `protected`). È possibile dichiarare una funzione o un'intera classe come `friend` di un'altra classe.

Le funzioni `friend` sono importanti dal punto di vista delle prestazioni. Qui vi proponiamo un esempio che illustra il funzionamento di una funzione `friend`. Nel resto del testo utilizzeremo le funzioni `friend` per effettuare l'overloading degli operatori e per creare classi iteratore. Gli oggetti di una classe iteratore servono a selezionare o a effettuare operazioni su elementi successivi di un oggetto di una classe container (*cfr. Sezione 7.9*). Nel Capitolo 8 vedremo che spesso bisognerà ricorrere alle funzioni `friend` se una funzione membro non può effettuare determinate operazioni.

Per dichiarare una funzione come `friend` di una classe, basta precedere con la parola chiave `friend` il suo prototipo nella definizione della classe. Per dichiarare la classe `ClassTwo` come `friend` della classe `ClassOne`, si scrive una dichiarazione di questo tipo

```
friend class ClassTwo;
```

all'interno della definizione di `ClassOne`.

Ingegneria del software 7.9

Anche se i prototipi delle funzioni `friend` *compaiono nella definizione di una classe, le funzioni* `friend` *non sono funzioni membro.*

Ingegneria del software 7.10

Le modalità di accesso `private`, `protected` *e* `public` *non sono rilevanti rispetto alle dichiarazioni* `friend`, *per cui queste dichiarazioni possono essere poste ovunque all'interno di una definizione di classe.*

Buona abitudine 7.2

È buona norma scrivere le dichiarazioni `friend` *immediatamente dopo l'intestazione della classe senza precederle con alcuno specificatore d'accesso.*

La relazione `friend` è una concessione, non una pretesa: perché la classe `B` sia `friend` della classe `A`, la classe `A` deve dichiarare esplicitamente che la classe `B` è `friend`. Inoltre questa relazione non è né simmetrica né transitiva: se la classe `A` è `friend` della classe `B`, e la classe `B` è `friend` della classe `C`, ciò *non* implica che la classe `B` sia `friend` della classe `A` (la relazione non è simmetrica) e non implica nemmeno che la classe `C` sia `friend` della classe `B` o che la classe `A` sia `friend` della classe `C` (la relazione non è transitiva).

Ingegneria del software 7.11

Alcuni fra i puristi della OOP pensano che la relazione `friend` *violi in qualche misura l'occultamento delle informazioni e depauperi l'approccio della programmazione orientata agli oggetti.*

La Figura 7.5 illustra la dichiarazione e l'uso di una funzione `friend`, `setX`, che serve per impostare `x`, che è un dato membro `private` della classe `Count`. Qui abbiamo seguito la convenzione di porre subito la dichiarazione `friend` nella dichiarazione di classe, anche prima delle funzioni membro `public`. Il programma in Figura 7.6 mostra i messaggi del compilatore quando viene invocata `cannotSetX`, funzione non `friend`, per modificare `x`. Le Figure 7.5 e 7.6 vogliono introdurre in un certo senso la "meccanica" delle funzioni `friend`, rimandiamo gli esempi pratici di uso delle funzioni `friend` ai prossimi capitoli.

```
1   // Fig. 7.5: fig07_05.cpp -  Le funzioni friend possono
2   // accedere ai dati membro private di una class.
3   #include <iostream.h>
4
5   // Classe Count modificata
6   class Count {
7      friend void setX( Count &, int ); // dichiarazione di funzione
8                                        // friend
9   public:
10     Count() { x = 0; }                // costruttore
11     void print() const { cout << x << endl; }  // output
12  private:
13     int x;  // dato membro
14  };
```

Figura 7.5 Le funzioni friend possono accedere ai dati private di una classe (continua)


```
15
16  // Può modificare il dato private di Count perché
17  // setX è dichiarata come funzione friend di Count
18  void setX( Count &c, int val )
19  {
20     c.x = val;  // consentito: setX è friend di Count
21  }
22
23  int main()
24  {
25     Count counter;
26
27     cout << "counter.x after instantiation: ";
28     counter.print();
29     cout << "counter.x after call to setX friend function: ";
30     setX( counter, 8 );  // imposta x con la funzione friend
31     counter.print();
32     return 0;
33  }
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Figura 7.5 Le funzioni friend possono accedere ai dati membro private di una classe.

Osservate che la funzione `setX` (linea 18) è una funzione isolata, in stile C, e non è membro della classe `Count`. Perciò quando viene invocata `setX` per l'oggetto `counter` utilizziamo l'istruzione della linea 30

```
setX( counter, 8 );  // imposta x con la funzione friend
```

che riceve `counter` come argomento, anziché utilizzare un handle (come il nome dell'oggetto) per chiamare la funzione, come in

```
counter.setX( 8 );
```

```
1   // Fig. 7.6: fig07_06.cpp
2   // Le funzioni non friend e non membro non possono accedere
3   // ai dati private di una class.
4   #include <iostream.h>
5
6   // Classe Count modificata
7   class Count {
8   public:
9      Count() { x = 0; }                          // costruttore
10     void print() const { cout << x << endl; }  // output
11  private:
12     int x;                                     // dato membro
13  };
```

Figura 7.6 Le funzioni non friend e non membro non possono accedere ai membri private di una classe (continua)

```
14
15    // La funzione tenta di modificare il dato private di Count,
16    // ma non può perché non è friend di Count.
17    void cannotSetX( Count &c, int val )
18    {
19       c.x = val;  // ERRORE: 'Count::x' non è accessibile
20    }
21
22    int main()
23    {
24       Count counter;
25
26       cannotSetX( counter, 3 ); // cannotSetX non è friend
27       return 0;
28    }
```

```
Compiling...
Fig07_06.cpp
Fig07_06.cpp(19) : error: 'x' :
cannot access private member declared in class 'Count'
```

Figura 7.6 Le funzioni non friend e non membro non possono accedere ai membri private di una classe.

Ingegneria del software 7.12

Il C++ è un linguaggio ibrido, per cui in uno stesso programma si possono trovare comunemente entrambi i tipi di chiamate di funzione, anche in uno stesso contesto: chiamate in stile C che passano dati primitivi o oggetti a funzioni e chiamate in stile C++ che inviano messaggi a oggetti.

È possibile specificare l'overloading di funzioni come `friend` di una classe: ogni ridefinizione che si vuole rendere `friend` della classe deve essere presente esplicitamente nella definizione della classe come `friend` di tale classe.

7.5 Il puntatore this

Ogni oggetto può accedere al proprio indirizzo tramite un puntatore di nome `this`. Il puntatore `this` di un oggetto non fa parte dell'oggetto, per cui non incide sull'occupazione di memoria dell'oggetto: in altre parole non ha alcuna influenza sul risultato di una operazione `sizeof` sull'oggetto. Possiamo affermare, piuttosto, che il compilatore passa il puntatore `this` a un oggetto come primo argomento implicito ad ogni chiamata di una funzione membro non statica (i membri statici sono discussi nella Sezione 7.7).

Il puntatore `this` viene usato implicitamente per riferire sia i dati che le funzioni membro di un oggetto, e può essere anche utilizzato esplicitamente. Il tipo di `this` dipende dal tipo dell'oggetto e dal fatto che la funzione membro in cui viene usato sia dichiarata come `const` o meno. In una funzione membro non constante della classe `Employee`, il tipo di `this` è `Employee * const` (puntatore constante a un oggetto `Employee`). In una funzione membro di `Employee`, invece, il tipo di `this` è `const Employee * const` (puntatore constante a un oggetto `Employee` constante).

Per il momento vi illustriamo soltanto un semplice esempio di puntatore `this` utilizzato esplicitamente; nel seguito di questo capitolo e nel Capitolo 8 vi mostreremo esempi più significativi dell'uso di `this`. Ogni funzione membro non statica può accedere al puntatore `this` dell'oggetto per cui è stata invocata.

Obiettivo efficienza 7.3

Per una questione di economia di memoria, esiste una sola copia di ogni funzione membro per ogni classe, e questa copia è invocata da ogni oggetto della classe. Ogni oggetto però ha la propria copia dei dati membro della classe.

La Figura 7.7 illustra il puntatore `this` utilizzato esplicitamente per consentire a una funzione membro della classe `Test` di visualizzare `x`, dato `private` di un oggetto `Test`.

```
1    // Fig. 7.7: fig07_07.cpp
2    // Uso del puntatore this per riferire gli oggetti membro.
3    #include <iostream.h>
4
5    class Test {
6    public:
7       Test( int = 0 );              // costruttore di default
8       void print() const;
9    private:
10      int x;
11   };
12
13   Test::Test( int a ) { x = a; }  // costruttore
14
15   void Test::print() const  // le ( ) che racchiudono *this sono
16                             // necessarie
17   {
18      cout << "        x = " << x
19           << "\n  this->x = " << this->x
20           << "\n(*this).x = " << ( *this ).x << endl;
21   }
22
23   int main()
24   {
25      Test testObject( 12 );
26
27      testObject.print();
28
29      return 0;
30   }
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Figura 7.7 Uso del puntatore this.

Perché possiate confrontare le notazioni, abbiamo scritto la funzione membro `print` in Figura 7.7 in modo da visualizzare `x` prima in modo diretto e, in seguito, abbiamo utilizzato due notazioni alternative per accedere a `x`: la prima fa uso dell'operatore freccia (`->`) sul puntatore `this`, la seconda dell'operatore punto (`.`) sul puntatore `this` dereferenziato.

Le parentesi che racchiudono `*this` sono necessarie quando si deve utilizzare l'operatore punto (`.`): infatti la precedenza dell'operatore punto è più alta di quella dell'operatore `*`. In mancanza di parentesi, l'espressione

```
*this.x
```

equivale all'espressione:

```
*( this.x )
```

che causa un errore di sintassi, perché l'operatore punto non si può utilizzare con un puntatore.

Errore tipico 7.7

Se tentate di utilizzare l'operatore di selezione del membro (.) con un puntatore a un oggetto commettete un errore di sintassi: l'operatore punto si può utilizzare soltanto con un oggetto o un riferimento a un oggetto.

Un uso interessante di `this` consiste nell'intercettare un autoassegnamento, che si verifica quando un oggetto che viene assegnato a se stesso. Come vedremo nel Capitolo 8, l'autoassegnamento può causare seri errori quando l'oggetto contiene puntatori ad aree di memoria allocate in modo dinamico.

Un altro uso di `this` è consentire chiamate a cascata a funzioni membro. In Figura 7.8 potete vedere la restituzione di un riferimento a un oggetto `Time`, per consentire chiamate a cascata a funzioni membro della classe `Time`. Le funzioni membro `setTime`, `setHour`, `setMinute` e `setSecond` restituiscono tutte `*this`, e dichiarano di restituire il tipo `Time &`.

```
1  // Fig. 7.8: time6.h
2  // Chiamate di funzioni membro a cascata.
3
4  // Dichiarazione della classe Time.
5  // Funzioni membro definite in time6.cpp
6  #ifndef TIME6_H
7  #define TIME6_H
8
9  class Time {
10 public:
11    Time( int = 0, int = 0, int = 0 );  // costruttore di default
12
13    // funzioni set
14    Time &setTime( int, int, int ); //imposta hour, minute, second
15    Time &setHour( int );     // imposta hour
16    Time &setMinute( int );   // imposta minute
17    Time &setSecond( int );   // imposta second
```

Figura 7.8 Chiamate a cascata a funzioni membro (continua).


```
18
19    // funzioni get (di norma dichiarate come const)
20    int getHour() const;     // restituisce hour
21    int getMinute() const;   // restituisce minute
22    int getSecond() const;   // restituisce second
23
24    // funzioni print (di norma dichiarate come const)
25    void printMilitary() const; //mostra l'ora in formato militare
26    void printStandard() const; //mostra l'ora in formato standard
27 private:
28    int hour;             // 0 - 23
29    int minute;           // 0 - 59
30    int second;           // 0 - 59
31 };
32
33 #endif

34 // Fig. 7.8: time.cpp
35 // Definizioni delle funzioni membro della classe Time.
36 #include "time6.h"
37 #include <iostream.h>
38
39 // Funzione costruttore per inizializzare i dati private.
40 // Chiama la funzione membro setTime per impostare le variabili.
41 // I valori di default sono 0 (cfr. la definizione di classe).
42 Time::Time( int hr, int min, int sec )
43    { setTime( hr, min, sec ); }
44
45 // Imposta i valori di hour, minute e second.
46 Time &Time::setTime( int h, int m, int s )
47 {
48    setHour( h );
49    setMinute( m );
50    setSecond( s );
51    return *this;   // consente le chiamate a cascata
52 }
53
54 // Imposta hour
55 Time &Time::setHour( int h )
56 {
57    hour = ( h >= 0 && h < 24 ) ? h : 0;
58
59    return *this;   // consente le chiamate a cascata
60 }
61
62 // Imposta minute
63 Time &Time::setMinute( int m )
```

Figura 7.8 Chiamate a cascata a funzioni membro (continua)

```
64  {
65     minute = ( m >= 0 && m < 60 ) ? m : 0;
66
67     return *this;    // consente le chiamate a cascata
68  }
69
70  // Imposta second
71  Time &Time::setSecond( int s )
72  {
73     second = ( s >= 0 && s < 60 ) ? s : 0;
74
75     return *this;    // consente le chiamate a cascata
76  }
77
78  // Restituisce hour
79  int Time::getHour() const { return hour; }
80
81  // Restituisce minute
82  int Time::getMinute() const { return minute; }
83
84  // Restituisce second
85  int Time::getSecond() const { return second; }
86
87  // Visualizza l'orario in formato militare: HH:MM:SS
88  void Time::printMilitary() const
89  {
90     cout << ( hour < 10 ? "0" : "" ) << hour << ":"
91          << ( minute < 10 ? "0" : "" ) << minute;
92  }
93
94  // Visualizza l'orario in formato standard: HH:MM:SS AM (or PM)
95  void Time::printStandard() const
96  {
97     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
98          << ":" << ( minute < 10 ? "0" : "" ) << minute
99          << ":" << ( second < 10 ? "0" : "" ) << second
100         << ( hour < 12 ? " AM" : " PM" );
101 }
102
103 // Fig. 7.8: fig07_08.cpp
104 // Chiamate di funzioni membro a cascata
105 // con il puntatore this
106 #include <iostream.h>
107 #include "time6.h"
108
109 int main()
110 {
111    Time t;
```

Figura 7.8 Chiamate a cascata a funzioni membro (continua)


```
112    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
113    cout << "Military time: ";
114    t.printMilitary();
115    cout << "\nStandard time: ";
116    t.printStandard();
117
118    cout << "\n\nNew standard time: ";
119    t.setTime( 20, 20, 20 ).printStandard();
120    cout << endl;
121
122    return 0;
123 }
```

```
Military time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Figura 7.8 Chiamate a cascata a funzioni membro.

Come è possibile che funzioni questa tecnica di restituire `*this` come riferimento? L'operatore punto (.) associa da sinistra a destra, per cui l'espressione

```
t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
```

prende in considerazione prima `t.setHour( 18 )` quindi restituisce un riferimento all'oggetto `t`. Il resto dell'espressione viene interpretato come

```
t.setMinute( 30 ).setSecond( 22 );
```

Viene eseguita la chiamata a `t.setMinute( 30 )`, che restituisce l'equivalente di `t`. Il resto dell'espressione viene interpretato come

```
t.setSecond( 22 );
```

Osservate le chiamate:

```
t.setTime( 20, 20, 20 ).printStandard();
```

Anch'esse utilizzano le chiamate a cascata. Queste chiamate devono comparire in quest'ordine nell'espressione, perché `printStandard`, così come è definita nella classe, non restituisce un riferimento a `t`. Se scrivete la chiamata a `printStandard` prima della chiamata a `setTime` commettete un errore di sintassi.

7.6 L'allocazione dinamica della memoria: gli operatori new e delete

Gli operatori `new` e `delete` consentono di allocare dinamicamente la memoria per tutti i tipi di dato, sia per quelli predefiniti che per quelli definiti dall'utente. Essi operano in un modo migliore di quanto non facciano le corrispondenti funzioni del C `malloc` e `free`.

Osservate questa linea di codice:

```
TypeName *typeNamePtr;
```

In C per creare dinamicamente un oggetto di tipo `TypeName`, si dovrebbe scrivere

```
typeNamePtr = malloc( sizeof( TypeName ) );
```

Questa notazione richiede necessariamente una chiamata di funzione (a `malloc`) e l'uso esplicito dell'operatore `sizeof`. Nelle versioni del C precedenti all'ANSI C, era necessario anche effettuare il cast del puntatore restituito dalla funzione `malloc` nel tipo `(TypeName *)`. Il difetto principale di `malloc` (che costringeva alla scrittura vista sopra) sta nel fatto che essa non prevede alcun metodo di inizializzazione del blocco di memoria che alloca. In C++, invece, basta scrivere

```
typeNamePtr = new TypeName;
```

L'operatore `new` crea automaticamente un oggetto allocando la giusta dimensione in memoria, chiama il costruttore dell'oggetto e restituisce un puntatore del tipo corretto. Se `new` non trova abbastanza spazio in memoria per allocare l'oggetto, esso restituisce un puntatore all'indirizzo `0`. Per liberare lo spazio allocato per questo oggetto in C++ basta utilizzare l'operatore `delete`, come segue:

```
delete typeNamePtr;
```

In C++ è consentito fornire un inizializzatore per un oggetto appena creato:

```
float *thingPtr = new float ( 3.14159 );
```

l'istruzione precedente inizializza un oggetto `float` appena creato con il valore `3.14159`.

In questo segmento di programma creiamo un array di interi di 10 elementi e lo assegniamo a `arrayPtr`:

```
int *arrayPtr = new int[ 10 ];
```

L'array in seguito può essere eliminato dalla memoria con l'istruzione

```
delete [] arrayPtr;
```

In modo duale all'operatore `new`, anche l'operatore `delete` non si limita a liberare lo spazio allocato in memoria, ma invoca preventivamente il distruttore dell'oggetto di cui si richiede l'eliminazione.

Errore tipico 7.8

Non mescolate le modalità di allocazione dinamica della memoria, cioè la coppia di operatori `new` e `delete` con la coppia di funzioni `malloc` e `free`, o commetterete errori logici. Lo spazio creato da `malloc` non può essere liberato da `delete`, e allo stesso modo gli oggetti creati con `new` non possono essere distrutti con `free`.

Errore tipico 7.9

Per gli array occorre utilizzare `delete []` anziché il semplice `delete`, o si verificheranno errori logici in fase di esecuzione. Per evitarli, tenete a mente che lo spazio creato per un intero array deve essere deallocato con l'operatore `delete []` mentre lo spazio creato per un elemento individuale deve essere deallocato con il semplice operatore `delete`.

Buona abitudine 7.3

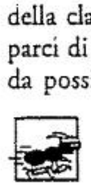
Dato che le funzionalità del C sono tutte presenti anche in C++, i programmi in C++ possono contenere chiamate a `malloc` e `free`, oltre a oggetti creati con `new` e distrutti con `delete`. Come regola generale, utilizzate unicamente `new` e `delete`.

7.7 I membri static di una classe

Ogni oggetto possiede una propria copia dei dati membro della sua classe ma, in alcuni casi, si potrebbe desiderare che una sola copia di un dato sia condivisa da tutti gli oggetti di una classe. In questo e in altri casi che citeremo si può utilizzare il modificatore `static` per dichiarare una variabile statica della classe.

Vogliamo farvi capire lo scopo dei dati `static`, e lo facciamo con un esempio tratto dal mondo dei videogiochi. Supponiamo di avere un videogioco con marziani (`Martian`) e altre terribili creature spaziali. Ogni `Martian` tende a farsi coraggio e ad attaccare le altre creature spaziali quando si rende conto che la flotta di `Martian` conta almeno 5 rappresentanti, mentre se ci sono meno di cinque `Martian` nelle vicinanze, ciascun `Martian` diventa codardo. Perciò ogni `Martian` deve sapere quanti sono i suoi colleghi nei paraggi: a questo punto possiamo dotare la classe `Martian` del dato membro `martianCount`, che indica il numero totale di `Martian` presenti. Facendo semplicemente in questo modo, ogni `Martian` avrà la sua copia separata del dato membro, e ogni volta che creiamo un nuovo `Martian` dovremo aggiornare tutte le copie di `martianCount` associate a ciascun `Martian`. Ciò costituisce uno spreco di spazio e di tempo, perché ci sono copie ridondanti dello stesso dato e devono essere aggiornate tutte a ogni creazione di un nuovo `Martian`.

Possiamo però decidere di dichiarare `martianCount` come membro `static`. Ciò rende `martianCount` un dato condiviso dall'intera classe. Ogni `Martian` può leggere `martianCount` come se fosse un proprio dato membro, ma esiste una sola copia del dato condivisa su tutti gli oggetti della classe. Oltre a un ovvio risparmio di spazio, considerate il notevole risparmio di tempo: faremo incrementare `martianCount` soltanto dal costruttore della classe `Martian`. Dato che ne esiste una sola copia, inoltre, non dobbiamo preoccuparci di incrementare i `martianCount` di tutti gli oggetti `Martian` e quindi ci cauteliamo da possibili errori.



Obiettivo efficienza 7.4

Un dato membro `static` consente di risparmiare spazio se serve una sola copia del dato per tutte le istanze di una data classe.

Anche se i dati membro `static` possono ricordare in qualche modo le variabili globali, tenete a mente che hanno scope di classe. I membri `static` possono essere `public`, `private` o `protected` e devono essere inizializzati una e una sola volta nello scope di file. A un dato membro `static` e `public` può accedere qualsiasi oggetto della classe, o si può accedere tramite il nome della classe e l'operatore binario di risoluzione dello scope. A un dato membro `static` e `private` (o `protected`) si deve accedere tramite le funzioni membro `public` della classe o tramite funzioni `friend` della classe.

Un membro `static` esiste anche quando non è stato ancora istanziato alcun oggetto della classe; in questo caso, se esso è `public`, per accedervi basta utilizzare il nome della classe e l'operatore binario di risoluzione dello scope (`::`) seguiti dal nome del dato membro. Se invece il dato membro statico è `private`, per accedervi anche in assenza di oggetti istanziati occorre una funzione membro definita come `static` e `public`. Questa deve essere chiamata specificando completamente, il nome della classe e l'operatore binario di risoluzione dello scope.

Il programma in Figura 7.9 mostra come accedere ai dati membro `static` e `private` tramite le funzioni membro `static` e `public`. Il dato membro `count` è inizializzato a zero nello scope a livello di file con l'istruzione

```
int Employee::count = 0;
```

Il dato membro `count` tiene il conto del numero di oggetti della classe `Employee` che *sono stati istanziati. Se esistono oggetti della classe `Employee`, il membro `count` può* essere riferito tramite qualsiasi funzione membro di un oggetto `Employee`: in questo esempio, `count` viene riferito sia dal costruttore che dal distruttore.

Errore tipico 7.10

Se includete la parola chiave `static` nella definizione di una variabile `static` di una classe nello scope di file, commettete un errore di sintassi.

```
// Fig. 7.9: employ1.h
// La classe Employee
#ifndef EMPLOY1_H
#define EMPLOY1_H

class Employee {
public:
   Employee( const char*, const char* );  // costruttore
   ~Employee();                           // distruttore
   const char *getFirstName() const;  // restituisce il nome
   const char *getLastName() const;   // restituisce il cognome

   // funzione membro static
   static int getCount();   // restituisce il numero di
                            // oggetti istanziati
private:
   char *firstName;
   char *lastName;

   // dato membro static
   static int count;  // numero di oggetti istanziati
};

#endif
// Fig. 7.9: employ1.cpp
// Definizioni delle funzioni membro della classe Employee
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "employ1.h"
```

Figura 7.9 Uso di un dato membro static che tiene il conto degli oggetti della classe istanziati (continua)


```
// Inizializza il dato membro static
int Employee::count = 0;

// Definisce la funzione membro static che
// restituisce il numero di oggetti Employee istanziati.
int Employee::getCount() { return count; }

// Costruttore che alloca spazio dinamicamente per il nome e
// il cognome e utilizza strcpy per copiarli nell'oggetto
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 );   // si assicura che la memoria sia
                               // stata effettivamente allocata
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );    // si assicura che la memoria sia
                               // stata effettivamente allocata
   strcpy( lastName, last );

   ++count;  // incrementa count
   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called." << endl;
}

// Distruttore che dealloca la memoria allocata dinamicamente
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;
   delete firstName;  // restituisce la memoria al sistema
   delete lastName;   // restituisce la memoria al sistema
   --count;  // decrementa count
}

// Restituisce il nome
const char *Employee::getFirstName() const
{
// la parola chiave const evita che il client possa modificare
// il dato private. Il client dovrebbe copiare la stringa
// restituita prima che il distruttore effettui la delete
// sulla memoria, per evitare un puntatore indefinito.
   return firstName;
}
```

Figura 7.9 Uso di un dato membro static che tiene il conto degli oggetti della classe istanziati (continua)

```
77
78  // Restituisce il cognome
79  const char *Employee::getLastName() const
80  {
81  // la parola chiave const evita che il client possa modificare
82  // il dato private. Il client dovrebbe copiare la stringa
83  // restituita prima che il distruttore effettui la delete
84  // sulla memoria, per evitare un puntatore indefinito.
85     return lastName;
86  }

87  // Fig. 7.9: fig07_09.cpp
88  // Programma di esempio per la classe Employee
89  #include <iostream.h>
90  #include "employ1.h"
91
92  int main()
93  {
94     cout << "Number of employees before instantiation is "
95          << Employee::getCount() << endl;    // utilizza il nome
96                                              // della classe
97
98     Employee *e1Ptr = new Employee( "Susan", "Baker" );
99     Employee *e2Ptr = new Employee( "Robert", "Jones" );
100
101    cout << "Number of employees after instantiation is "
102         << e1Ptr->getCount();
103
104    cout << "\n\nEmployee 1: "
105         << e1Ptr->getFirstName()
106         << " " << e1Ptr->getLastName()
107         << "\nEmployee 2: "
108         << e2Ptr->getFirstName()
109         << " " << e2Ptr->getLastName() << "\n\n";
110
111    delete e1Ptr;   // restituisce la memoria al sistema
112    e1Ptr = 0;
113    delete e2Ptr;   // restituisce la memoria al sistema
114    e2Ptr = 0;
115
116    cout << "Number of employees after deletion is "
117         << Employee::getCount() << endl;
118
119    return 0;
120 }
```

Figura 7.9 Uso di un dato membro static che tiene il conto degli oggetti della classe istanziati (continua)


```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

Figura 7.9 Uso di un dato membro static che tiene il conto degli oggetti della classe istanziati.

Se non esistono (ancora) oggetti della classe **Employee**, è ancora possibile accedere al membro **count**, ma bisogna farlo tramite una chiamata alla funzione membro statica **getCount**, come segue:

```
Employee::getCount()
```

In questo esempio la funzione **getCount** serve a determinare il numero di oggetti **Employee** istanziati in un dato momento. Osservate che se non è stato istanziato alcun oggetto nel programma, viene chiamata la funzione **Employee::getCount()**.

Invece, se ci sono oggetti istanziati, si può chiamare la funzione **getCount** per mezzo di un oggetto, come mostrano le linee 101 e 102 del codice

```
cout << "Number of employees after instantiation is "
     << e1Ptr->getCount();
```

In questa istruzione avremmo potuto scrivere naturalmente anche **e2Ptr->getCount()** o **Employee::getCount()**: il risultato sarebbe stato identico.

Ingegneria del software 7.13

È buona norma invocare le funzioni membro **static** *per mezzo del nome della classe e non attraverso gli oggetti della classe.*

Una funzione membro può essere dichiarata come **static** se non accede a dati e funzioni membro non **static**. A differenza delle funzioni membro non **static**, una funzione membro **static** non ha un puntatore **this** perché i dati e le funzioni membro **static** esistono indipendentemente dagli oggetti della classe.

Errore tipico 7.11

Se riferite il puntatore **this** *in una funzione membro* **static** *commettete un errore di sintassi.*

Errore tipico 7.12

Se dichiarate **const** *una funzione membro* **static** *commettete un errore di sintassi.*

Ingegneria del software 7.14

I dati e le funzioni membro `static` *di una classe esistono indipendentemente dalle istanze degli oggetti di tale classe: essi possono essere utilizzati anche se non è stato istanziato alcun oggetto.*

Alle linee 98 e 99 il programma alloca dinamicamente due oggetti `Employee` con l'operatore `new`. Per ogni oggetto `Employee` allocato viene chiamato il costruttore. Con la `delete` delle linee 111 e 113, che dealloca i due oggetti `Employee`, viene chiamato il distruttore per ciascuno di essi.

Buona abitudine 7.4

Dopo aver utilizzato `delete` *per deallocare la memoria degli oggetti, impostate il puntatore che riferiva l'area di memoria a* `0`*. In questo modo disconnetterete il puntatore da quest'area di memoria.*

Osservate l'uso di `assert` nel costruttore di `Employee`. La macro `assert` è definita nel file di intestazione `assert.h` e verifica il valore di una condizione. Se il valore è `false`, la macro `assert` causa un messaggio di errore e chiama la funzione `abort` (nel file di intestazione delle utilità generali `stdlib.h`) per terminare l'esecuzione del programma. Questa macro costituisce un valido aiuto per il debugging, nei casi in cui occorra verificare che una variabile abbia un valore significativo. In questo programma, la macro `assert` determina se l'operatore `new` sia stato in grado di soddisfare la richiesta di allocare dinamicamente spazio in memoria. Per esempio, nel costruttore di `Employee`, la seguente linea (detta anche *asserzione*):

```
assert( firstName != 0 );
```

effettua un test sul puntatore `firstName` per determinare se è uguale a `0`. Se la condizione risulta `true`, il programma continua l'esecuzione indisturbato. Se la condizione è `false`, viene visualizzato un messaggio di errore contenente il numero di linea, la condizione verificata e il nome del file in cui compare l'asserzione, e il programma termina. In questo modo il programmatore può studiare quest'area di codice per scoprire di che errore si tratta. Nel Capitolo 1 del volume Tecniche Avanzate introdurremo un metodo migliore per scoprire e correggere gli errori che si presentano durante l'esecuzione dei programmi.

Non è necessario eliminare manualmente tutte le asserzioni a fine debugging. Quando non se ne ha più bisogno basta inserire la linea

```
#define NDEBUG
```

all'inizio del file del programma. Il preprocessore ignorerà tutte le asserzioni, risparmiandovi l'onere di eliminarle una per una.

Se osservate, le implementazioni delle funzioni `getFirstName` e `getLastName` restituiscono al client della classe dei puntatori a caratteri constanti. In questa implementazione, se il client vuole tenere una copia del nome e del cognome, deve prendersi la responsabilità di copiare la memoria allocata dinamicamente nell'oggetto `Employee`, dopo aver ottenuto il puntatore dall'oggetto. Ad ogni modo possiamo anche implementare `getFirstName` e `getLastName` in modo tale che il client debba passare un array di caratteri e la sua dimensione a ogni funzione. Quindi la funzione dovrebbe copiare il nome o il cognome nell'array di caratteri fornito dal client.

7.8 Due concetti importanti: astrazione dei dati e occultamento delle informazioni

Le classi normalmente nascondono i dettagli della loro implementazione ai propri client: è questa la sostanza del principio di occultamento delle informazioni. Adesso consideriamo un esempio di occultamento delle informazioni, la struttura dati detta *pila*.

Per visualizzare una pila pensate a una pila di piatti. Quando mettete un altro piatto su una pila, dovete metterlo su tutti gli altri già presenti (effettuate quella che viene chiamata una *push*), e quando ne prendete uno dalla pila non potete che prendere l'ultimo piatto riposto (effettuate una *pop*). Le pile sono note come strutture dati *LIFO*, dall'inglese *last-in, first-out*, l'ultimo elemento immesso nella pila è anche il primo a essere recuperato.

È possibile creare una classe che implementi la pila e nasconderne l'implementazione ai client. Le pile si implementano facilmente tramite gli array (o le liste concatenate: cfr. il Capitolo 3 del volume Tecniche Avanzate). Il client di una classe pila non ha bisogno di conoscere l'implementazione della pila: l'unica cosa di cui ha bisogno è che le operazioni di push/pop rispecchino lo schema *last-in, first-out*. Se descriviamo la funzionalità di una classe indipendentemente dalla sua implementazione possiamo parlare di *astrazione dei dati*: le classi del C++ servono proprio a definire *tipi di dati astratti* (in inglese *Abstract Data Type* o *ADT*). Anche se gli utenti possono conoscere, sia pur accidentalmente, i dettagli di implementazione di una classe, non dovrebbero far dipendere il codice dei loro programmi da questi dettagli. Quello che intendiamo è che l'implementazione di una classe particolare, come può essere quella che implementa la pila e il suo meccanismo LIFO, deve poter essere sostituita di sana pianta magari con una migliore, senza influenzare il resto del sistema, a patto che l'interfaccia pubblica della classe resti inalterata.

Lo scopo dei linguaggi ad alto livello è creare una visione delle cose conveniente per i programmatori. Non esiste "la" visione delle cose, accettata universalmente e indiscussa, e questo è anche il motivo per cui esistono così tanti linguaggi di programmazione. La programmazione orientata agli oggetti del C++ presenta una visione alternativa.

La maggior parte dei linguaggi di programmazione è orientata all'azione: i dati esistono in quanto supporto alle azioni che devono essere intraprese. In un certo senso i dati sono visti come qualcosa di "meno interessante" delle azioni, sono espressi in modo "grezzo" ed è previsto soltanto un piccolo insieme di tipi di dato. Inoltre in questa categoria di linguaggi risulta piuttosto difficile creare nuovi tipi di dato.

Questa situazione è quasi rovesciata nella programmazione orientata agli oggetti e, dunque, in C++. L'attività principale in C++ è la creazione di nuovi tipi di dato (classi) e l'espressione delle interazioni che esistono tra di essi. Per seguire questa direzione, la comunità informatica ha dovuto formalizzare alcuni concetti sui dati, in particolare per ciò che riguarda il concetto di tipo di dato astratto. Gli ADT ricevono tanta attenzione al giorno d'oggi quanta ne ha ricevuta la programmazione strutturata nell'ultimo ventennio. Gli ADT non sostituiscono la programmazione strutturata; essi ne costituiscono una ulteriore formalizzazione destinata a migliorare la qualità del software.

Che cos'è un tipo di dato astratto? Pensate al tipo di dato predefinito `int`. Vi verrà in mente il concetto di intero in matematica, ma l'`int` di un computer non è precisamente uguale a un intero della matematica. In particolare, gli `int` dei computer normalmente

sono di dimensioni piuttosto limitate. Per esempio, `int` su una macchina a 32 bit è limitato all'incirca all'intervallo tra –2 miliardi e +2 miliardi. Se il risultato di un calcolo cade fuori di questo intervallo si ha un errore di "overflow" e il computer risponde in una qualche maniera, che varia caso per caso: è anche possibile che riporti "silenziosamente" un risultato scorretto. Con gli interi matematici questo problema non esiste. Quindi in realtà il concetto di `int` è soltanto un'approssimazione al concetto di intero del mondo reale. Lo stesso dicasi per il tipo `float`.

Persino `char` è un'approssimazione: i valori `char` sono normalmente schemi di 8 bit composti da zero e uno. Questi schemi non hanno molto a che fare con i caratteri che rappresentano, come la `Z` maiuscola, la `z` minuscola, il segno di dollaro (`$`), una cifra (`5`) e così via. I valori del tipo `char` della maggior parte dei computer sono piuttosto limitati in confronto al numero di caratteri possibili. Il set di caratteri a 7 bit ASCII prevede 128 valori diversi per i caratteri: questo set è decisamente inadeguato a rappresentare per esempio i caratteri giapponesi e cinesi, che sono migliaia. Il punto è che persino i tipi di dato predefiniti "forniti di serie" insieme con i linguaggi ad alto livello come il C++ sono soltanto delle approssimazioni o dei modelli limitati di concetti e comportamenti del mondo reale. Fino a questo punto avete dato per scontato `int`, ma adesso ne sapete abbastanza per considerare le cose in prospettiva. I tipi di dato come `int`, `float` e `char` sono già esempi di tipi di dato astratto. Essenzialmente sono modi di rappresentare i concetti del mondo reale a un livello di precisione accettabile.

Il concetto di tipo di dato astratto richiama altre due nozioni, quella di rappresentazione di un dato e di operazioni consentite su tale dato. Per esempio, il concetto di `int` definisce l'addizione, la sottrazione, la moltiplicazione, la divisione e il modulo in C++, ma la divisione per zero resta indefinita: inoltre il modo in cui sono effettuate tutte queste operazioni può variare da macchina a macchina, a seconda della lunghezza della parola, che costituisce l'unità fondamentale di memorizzazione del computer. Un altro esempio si ha con il concetto di intero negativo, per il quale sono chiare operazioni e rappresentazione, ma resta indefinita l'estrazione della radice quadrata. In C++ i programmatori implementano i tipi di dato astratti tramite le classi. Nel Capitolo 1 del volume Tecniche Avanzate *creeremo una nostra classe pila, e nel Capitolo 8 dello stesso volume, studieremo la* classe `stack` (pila) della libreria standard.

7.8.1 Il tipo di dato astratto "array"

Abbiamo già parlato degli array nel Capitolo 4. Un array non è molto diverso da un puntatore a un'area di memoria. Questa definizione primitiva è abbastanza accettabile per effettuare tutte le operazioni fondamentali sugli array, senza pretendere troppo. Tuttavia ci sono alcune operazioni che sarebbe comodo poter effettuare sugli array, ma che non sono predefinite in C++. Niente paura, grazie alle classi è possibile sviluppare l'array come *ADT*, in modo da creare una versione *preferibile* agli array predefiniti. Una classe che implementa gli array può prevedere nuove funzionalità interessanti come ad esempio:

- Controllo della validità degli indici.
- Libera scelta sull'intervallo degli indici, senza dover iniziare per forza da 0.
- Assegnamento di due array.

- Confronto tra due array.
- Input e output di array.
- Array che "conoscono" le proprie dimensioni.

Nel Capitolo 8 creeremo la nostra classe array, mentre nel Capitolo 9 del volume Tecniche Avanzate ne studieremo l'implementazione della libreria standard (`vector`).

Ingegneria del software 7.15

Grazie alle classi potete creare nuovi tipi di dato e potete progettarli in modo che possiate utilizzarli con la stessa comodità dei tipi predefiniti. Per questo motivo diciamo che il C++ è un linguaggio estensibile. Tuttavia, sebbene sia facile estendere le funzionalità del linguaggio con questi meccanismi, la base del linguaggio non si può modificare.

Le classi C++ possono essere di proprietà di un individuo, di un gruppo di lavoro o di aziende informatiche e possono essere raccolte in librerie per essere distribuite sul mercato. Chi impara il C++ oggi potrà approfittare delle nuove possibilità offerte dal rapido sviluppo di componenti software e di librerie sempre più ricche di funzionalità.

7.8.2 Il tipo di dato astratto "stringa"

Il C++ è stato progettato intenzionalmente come linguaggio essenziale, che fornisce soltanto gli strumenti di base per estenderlo e creare sistemi diversi. Inoltre è stato progettato per massimizzare l'efficienza del codice: è per questo che il C++ è adatto sia per le applicazioni che per i sistemi operativi (ed è proprio in questo caso che si possono apprezzarne le elevate prestazioni). Non sarebbe stato certamente difficile includere un tipo di dato "stringa" tra quelli predefiniti, ma il linguaggio fu progettato per includere invece il meccanismo dei dati astratti, tramite i quali sarebbe stato possibile creare in seguito un tipo "stringa". Nel Capitolo 8 creeremo il nostro tipo di dato stringa come ADT. Il draft dello standard ANSI/ISO prevede una classe `string`, di cui parleremo in dettaglio nel Capitolo 8 del volume Tecniche Avanzate.

7.8.3 Il tipo di dato astratto "coda"

Chi di noi di tanto in tanto non deve mettersi in coda? Una coda informatica è simile a quella che dobbiamo affrontare alla cassa di un supermercato, dal benzinaio, alla fermata dell'autobus o all'ingresso dell'autostrada. I computer si servono spesso delle code internamente, per cui è utile scrivere programmi che simulino le code e il loro funzionamento.

La coda è un buon esempio di tipo di dato astratto: essa offre un comportamento ben definito ai suoi client. Essi mettono gli elementi in coda uno dopo l'altro (accodamento) e li ritirano su richiesta uno alla volta, nell'ordine in cui sono stati accodati. Teoricamente una coda può allungarsi all'infinito. Una coda reale, naturalmente ha sempre un inizio e una fine. Gli elementi di una coda seguono il meccanismo FIFO, dall'inglese *first-in, first-out*: il primo elemento inserito è il primo a essere recuperato dalla coda.

La coda prevede una rappresentazione interna che tiene traccia dell'elemento da servire in un certo momento, e offre una serie di servizi ai client, ovvero l'accodamento e il recupero degli elementi. I client non devono occuparsi di come è implementata la coda, vogliono soltanto che si comporti "secondo le regole". Quando un client accoda un nuovo

elemento, la coda dovrebbe accettarlo e porlo in una struttura interna che funziona secondo lo schema FIFO. Quando il client desidera l'elemento successivo in coda, la coda dovrebbe eliminarlo dalla sua rappresentazione interna e consegnarlo al mondo esterno, cioè al client, secondo l'ordine FIFO: l'elemento consegnato è quello che è rimasto più a lungo in coda.

L'implementazione della coda come ADT garantisce l'integrità della struttura dati interna non consentendo ai client di manipolare direttamente questa struttura dati. Soltanto la coda (come ADT) può accedere ai suoi dati interni; in questo modo i client possono effettuare soltanto le operazioni consentite sulla rappresentazione dei dati: le operazioni che non sono presenti nell'interfaccia pubblica dell'ADT sono respinte dall'ADT nella maniera più opportuna, ovvero visualizzando un messaggio di errore, causando il termine dell'esecuzione del programma o semplicemente ignorando la richiesta del client.

Nel Capitolo 4 del volume Tecniche Avanzate creeremo la nostra classe coda, mentre nel Capitolo 9 del volume Tecniche Avanzate studieremo la classe **queue** (coda) della libreria standard.

7.9 Le classi container e gli iteratori

Tra i tipi di classi più comuni troviamo le *classi container* (o *collezioni*), che sono classi destinate a contenere collezioni di oggetti. Le classi container permettono generalmente operazioni sugli elementi quali l'inserimento, l'eliminazione, la ricerca, l'ordinamento, la verifica che un elemento sia membro di una data classe e così via. Array, pile, code, alberi e liste concatenate sono tutti esempi di classi container. Abbiamo studiato gli array nel Capitolo 4, e studieremo le altre nei Capitoli 4 e 9 del volume Tecniche Avanzate.

Di solito alle classi container si associano *oggetti iteratori*, o più semplicemente *iteratori*. Un iteratore è un oggetto che restituisce l'elemento successivo di una collezione o che effettua un'operazione su di esso. Prevedendo un iteratore per una classe, si può ottenere l'elemento successivo di una collezione utilizzando una notazione notevolmente semplificata. Gli iteratori normalmente sono definiti come **friend** delle classi, per motivi di efficienza: in questo modo possono accedere direttamente ai dati **private** delle classi che "iterano". Allo stesso modo in cui un libro condiviso tra più persone può contenere diversi segnalibri in uno stesso momento, una classe container può avere diversi iteratori che operano su di essa nello stesso momento. Ogni iteratore tiene traccia della propria "posizione". Torneremo a parlare in maniera approfondita degli iteratori e delle classi container nel Capitolo 9 del volume Tecniche Avanzate.

7.10 Le classi proxy

È desiderabile nascondere i dettagli dell'implementazione di una classe, per evitare che estranei accedano alle informazioni (tra cui i dati **private**) e alla logica del programma, che devono rimanere proprietarie. Perciò potete decidere di fornire ai client della vostra classe una classe *proxy*, che conosce soltanto l'interfaccia pubblica della classe ma consente ai client di utilizzare i servizi della classe originaria senza conoscere i dettagli dell'implementazione della vostra classe.

L'implementazione di una classe proxy richiede numerosi passi (Figura 7.10). Per prima cosa creiamo i file che contengono la definizione e l'implementazione della classe che vogliamo proteggere. La classe del nostro esempio, **Implementation**, è implementation.h. La classe proxy **Interface** è interface.h e interface.cpp, e il programma di esempio, con relativo output, è fig.07_10.cpp.

```
1   // Fig. 7.10: implementation.h
2   // File di intestazione per la classe Implementation
3
4   class Implementation {
5      public:
6         Implementation( int v ) { value = v; }
7         void setValue( int v ) { value = v; }
8         int getValue() const { return value; }
9
10     private:
11        int value;
12  };

13  // Fig. 7.10: interface.h
14  // File di intestazione per interface.cpp
15  class Implementation;   // dichiarazione anticipata
16
17  class Interface {
18   - public:
19        Interface( int );
20        void setValue( int );  // stessa interfaccia pubblica della
21        int getValue() const;  // classe Implementation
22    private:
23        Implementation *ptr;   // richiede la
24                               // dichiarazione anticipata
25  };

26  // Fig. 7.10: interface.cpp
27  // Definizione della classe Interface.
28  #include "interface.h"
29  #include "implementation.h"
30
31  Interface::Interface( int v )
32     : ptr ( new Implementation( v ) ) { }
33
34  // chiama la funzione setValue di Implementation
35  void Interface::setValue( int v ) { ptr->setValue( v ); }
36
37  // chiama la funzione getValue di Implementation
38  int Interface::getValue() const { return ptr->getValue(); }
```

Figura 7.10 Implementazione di una classe proxy (continua)

```
39  // Fig. 7.10: fig07_10.cpp - Occultamento dei dati
40  // private di una classe tramite una classe proxy.
41  #include <iostream.h>
42  #include "interface.h"
43
44  int main()
45  {
46     Interface i( 5 );
47
48     cout << "Interface contains: " << i.getValue()
49          << " before setValue" << endl;
50     i.setValue( 10 );
51     cout << "Interface contains: " << i.getValue()
52          << " after setValue" << endl;
53     return 0;
54  }
```

```
Interface contains: 5 before setVal
Interface contains: 10 after setVal
```

Figura 7.10 Implementazione di una classe proxy.

La classe `Implementation` (implementation.h) ha un solo dato membro `private` di nome `value` (che è il dato che vogliamo nascondere allo sguardo dei client), un costruttore che inizializza `value` e le funzioni `setValue` e `getValue`.

Abbiamo scritto la definizione della classe proxy con interfaccia pubblica identica a quella della classe `Implementation`. L'unico membro `private` della classe proxy è un puntatore a un oggetto della classe `Implementation`. Utilizzando un puntatore in questo modo possiamo nascondere i dettagli dell'implementazione di `Implementation` al client.

La classe `Interface` (interface.h) è la classe proxy per `Implementation`. Osservate che l'unico riferimento alla classe proprietaria `Implementation` all'interno della classe `Interface` è un puntatore (linea 23). Quando la definizione di una classe (come quella di `Interface`) utilizza soltanto un puntatore a un'altra classe (come quello a `Implementation`), non è necessario includere il file di intestazione dell'altra classe con `#include`, cosa che naturalmente rivelerebbe i dati `private` di tale classe. Si può invece dichiarare semplicemente l'altra classe come tipo di dato con una *dichiarazione anticipata* (linea 15), prima che il tipo di dato sia utilizzato nel file. Il file dell'implementazione contenente le funzioni membro della classe proxy `Interface` (interface.cpp) è l'unico file a includere `implementation.h`, che contiene la classe `Implementation`. Il file `interface.cpp` viene fornito al client come file oggetto precompilato insieme con il file di intestazione `interface.h`, che include i prototipi di funzione dei servizi forniti dalla classe proxy. Dato che il client è in possesso soltanto della versione precompilata di `interface.cpp`, non è in grado di vedere le interazioni tra la classe proxy e la classe proprietaria.

Il programma di esempio in Figura 7.10 (fig07_10.cpp) fa uso della classe `Interface`. Osservate che in `main` viene incluso soltanto il file di intestazione della classe `Interface`, mentre non c'è alcun riferimento all'esistenza di una classe distinta di nome `Implementation`. In questo modo il client non vedrà mai i dati `private` della classe `Implementation`.

7.11 Pensare in termini di oggetti: l'implementazione delle classi del simulatore [progetto opzionale]

Nelle sezioni "Pensare in termini di oggetti" dei Capitoli dal 2 al 5 abbiamo elaborato il progetto del simulatore, e in quella del Capitolo 6 abbiamo iniziato la sua programmazione in C++. Nel corso di questo capitolo abbiamo illustrato altre funzionalità utili allo scopo di completarne l'implementazione. Abbiamo trattato la creazione e la distruzione dinamica degli oggetti per mezzo degli operatori `new` e `delete`, ed abbiamo introdotto il concetto di *composizione, una funzionalità che ci consente di creare classi che contengono oggetti di* altre classi. Grazie ad essa potremo creare la classe `Building` che conterrà un oggetto `Scheduler`, un oggetto `Clock`, un oggetto `Elevator` e due oggetti `Floor`; potremo creare poi la classe `Elevator` che conterrà a sua volta un oggetto delle classi `ElevatorButton`, `Door` e `Bell`; infine potremo creare la classe `Floor` che conterrà oggetti `FloorButton` e `Light`. Nel corso del capitolo abbiamo anche visto come utilizzare i modificatori `static` e `const` per le classi membro e la sintassi degli inizializzatori di membro nei costruttori. Nel corso di questa sezione proseguiremo, nell'implementazione del sistema ascensore in C++ utilizzando queste tecniche, e alla fine di questa sezione presenteremo il programma completo (consistente di circa 1000 linee di codice) e una sua analisi dettagliata. Nel Capitolo 9 completeremo il simulatore sfruttando il concetto di ereditarietà, e presenteremo soltanto il codice da integrare all'implementazione proposta in questo capitolo.

7.11.1 Una panoramica dell'implementazione

La simulazione dell'ascensore è controllata da un oggetto della classe `Building` che contiene due oggetti della classe `Floor` e uno delle classi `Elevator`, `Clock` e `Scheduler`. Abbiamo descritto questa relazione di composizione nel diagramma delle classi UML (Figura 2.44) del Capitolo 2. Ricordiamo che il timer (`Clock`) ha il compito di cronometrare la simulazione ed è incrementato dall'edificio ad intervalli regolari. Il meccanismo di gestione degli arrivi (`Scheduler`), invece, decide l'arrivo delle persone su ogni piano.

Dopo ogni istante, misurato dal timer, l'edificio passa l'orario corrente all'oggetto `Scheduler` attraverso la funzione membro `processTime`. Questo oggetto controlla se l'istante corrente è uguale a quello previsto per l'arrivo di una persona ad un dato piano, ed in caso affermativo, verifica se ci sia già una persona in attesa al piano in questione chiamando la funzione membro `isOccupied` della classe `Floor`. Se il risultato di questa chiamata è `true`, significa che sul piano è già presente una persona in attesa, per cui il meccanismo di gestione degli arrivi invoca la propria funzione `delayArrival` per ritardare di un istante temporale l'arrivo della persona successiva.

Se il piano è libero, cioè se `isOccupied` restituisce `false`, il programmatore crea un nuovo oggetto della classe `Person`, che accede al piano in questione e preme il pulsante esterno invocando la funzione membro `pressButton` della classe `FloorButton`. Infine, quest'ultimo chiama la cabina dell'ascensore al piano sul quale è situato invocando il metodo `summonElevator` della classe `Elevator`.

Dopo aver passato l'istante corrente al meccanismo di gestione degli arrivi, l'edificio comunica questa informazione anche all'ascensore. Esso, a sua volta controlla il proprio stato, che può essere "in corsa" o "fermo". Nel caso l'ascensore sia in corsa, ma il suo

arrivo al piano non sia previsto per l'orario corrente, esso si limita a inviare in output sullo schermo la direzione del suo moto. Qualora, invece, l'ascensore sia in corsa e il suo arrivo ad un piano sia previsto per l'istante corrente, esso si ferma, rilascia il pulsante interno, suona il campanello e notifica il suo arrivo al piano invocando la funzione membro `elevatorArrived` della classe `Floor`. In risposta il piano rilascia il pulsante esterno e accende la spia di presenza. L'ascensore apre quindi la porta, consentendo al passeggero di uscire e all'eventuale persona in attesa al piano di entrare, quindi chiude nuovamente la porta e determina se deve servire l'altro piano. In tal caso l'ascensore inizia una nuova corsa verso l'altro piano ponendosi nuovamente nello stato "in corsa".

Nel caso, invece, in cui l'ascensore sia fermo quando riceve dall'edificio l'istante corrente della simulazione, esso determina per prima cosa quale piano servire. Se si tratta del piano corrente, cioè una persona ha premuto il pulsante esterno su tale piano, l'ascensore suona il campanello, notifica al piano la sua presenza e apre la porta. A questo punto, la persona al piano può entrare e premere il pulsante interno per andare all'altro piano. Se, invece, una persona ha premuto il pulsante esterno all'altro piano, l'ascensore si pone nello stato "in corsa" effettuando una corsa verso tale piano.

7.11.2 Implementazione del simulatore

Nelle sezioni "Pensare in termini di oggetti" dei capitoli precedenti abbiamo raccolto diverse informazioni sul nostro sistema che abbiamo utilizzato per progettare la nostra simulazione orientata agli oggetti rappresentandole utilizzando UML. Nel corso di questi capitoli, inoltre, abbiamo discusso tutti gli aspetti della programmazione orientata agli oggetti in C++ necessari per implementare il nostro simulatore. Dunque, siamo pronti a partire con l'implementazione in C++ del simulatore che sarà presentata nel resto di questa sezione assieme ad una sua analisi dettagliata.

La funzione `main` del programma (Figura 7.11) chiede innanzitutto all'utente di digitare la durata totale della simulazione (linee 11-12). La chiamata a `cin.ignore` della linea 13 ordina allo stream `cin` di ignorare il carattere di Invio che l'utente digita dopo il numero intero durante l'esecuzione. Il programma crea quindi l'oggetto `building` (linea 15) e invoca la sua funzione membro `runSimulation`, passandole come parametro la durata specificata dall'utente (linea 19). Il programma visualizza inoltre dei messaggi che indicano all'utente quando la simulazione comincia (linea 17) e finisce (linea 20).

```
1  // main.cpp
2  // Sezione principale del programma di simulazione.
3  #include <iostream.h>
4
5  #include "building.h"
6
7  int main()
8  {
9        int duration; // durata della simulazione in secondi
10
11       cout << "Enter run time: ";
12       cin >> duration;
```

Figura 7.11 Sezione principale del programma di simulazione (continua)


```
13       cin.ignore(); // ignora il carattere Invio
14
15       Building building; // crea l'edificio
16
17       cout << endl << "*** ELEVATOR SIMULATION BEGINS ***"
18            << endl << endl;
19       building.runSimulation( duration ); //inizia la simulazione
20       cout << "*** ELEVATOR SIMULATION ENDS ***" << endl;
21
22       return 0;
23 }
```

Figura 7.11 Sezione principale del programma di simulazione.

Secondo quanto abbiamo descritto nel diagramma delle classi di Figura 2.44, `Building` è composto da oggetti di diverse altre classi ed il file di intestazione di `Building` (Figura 7.12) riflette questo aspetto nelle linee 42–46. La classe `Building` è, infatti, composta da due oggetti `Floor` (`floor1` e `floor2`), un oggetto `Elevator` (`elevator`), un oggetto `Clock` (`clock`) e un oggetto `Scheduler` (`scheduler`).

```
24 // building.h
25 // Definizione della classe Building.
26 #ifndef BUILDING_H
27 #define BUILDING_H
28
29 #include "elevator.h"
30 #include "floor.h"
31 #include "clock.h"
32 #include "scheduler.h"
33
34 class Building {
35
36 public:
37       Building(); // costruttore
38       ~Building(); // distruttore
39       void runSimulation( int ); // esegue la simulazione per il
40                                  // tempo specificato
41 private:
42       Floor floor1; // oggetto "primo piano"
43       Floor floor2; // oggetto "secondo piano"
44       Elevator elevator; // oggetto "ascensore"
45       Clock clock; // oggetto "timer"
46       Scheduler scheduler; // oggetto che gestisce gli arrivi
47 };
48
49 #endif // BUILDING_H
```

Figura 7.12 File di intestazione della classe Building.

Il file di implementazione della classe `Building` è riportato in Figura 7.13. Il costruttore si trova alle linee 56-61; nell'elenco degli inizializzatori di membro (linee 57–60) vengono

chiamati i costruttori di molti degli oggetti che compongono **Building**, naturalmente con gli argomenti appropriati. **FLOOR1** e **FLOOR2** (linee 57–58) sono costanti definite nella classe **Floor** (linee 785–786) e indicano, rispettivamente, il primo ed il secondo piano.

```
50  // building.cpp
51  // Definizione delle funzioni membro della classe Building.
52  #include <iostream.h>
53
54  #include "building.h"
55
56  Building::Building() // costruttore
57        : floor1( Floor::FLOOR1, elevator ),
58        floor2( Floor::FLOOR2, elevator ),
59        elevator( floor1, floor2 ),
60        scheduler( floor1, floor2 )
61  { cout << "building created" << endl; }
62
63  Building::~Building() // distruttore
64  { cout << "building destroyed" << endl; }
65
66  // controlla la simulazione
67  void Building::runSimulation( int totalTime )
68  {
69        int currentTime = 0;
70
71        while ( currentTime < totalTime ) {
72              clock.tick();
73              currentTime = clock.getTime();
74              cout << "TIME: " << currentTime << endl;
75              scheduler.processTime( currentTime );
76              elevator.processTime( currentTime );
77              cin.get();//arresta l'output per agevolarne la lettura
78        }
79  }
```

Figura 7.13 File di implementazione della classe Building.

La funzionalità principale della classe **Building** è **runSimulation** (linee 66–79), ed è costituita da un ciclo che si interrompe quando è trascorsa la quantità di tempo specificata. Ad ogni iterazione, **building** ordina a **clock** di incrementare l'istante corrente, inviando a **clock** il messaggio **tick** (linea 72). A quel punto **building** interroga **clock** per conoscere il nuovo valore temporale, chiamando la funzione membro **getTime** (linea 73), e lo memorizza nella variabile**currentTime**. Tale valore viene quindi inviato tramite il messaggio **processTime** a **scheduler** ed **elevator**, rispettivamente alle linee 75 e 76. Infine troviamo una chiamata a **cin.get** (linea 77), che consente all'utente di interrompere temporaneamente lo scorrimento dell'output per poter leggere lo stato della simulazione nell'istante successivo, prima di premere il tasto Invio che lo fa ripartire.

Clock è una classe semplice, non essendo composta di altri oggetti. Il file di intestazione della classe **Clock** è riportato in Figura 7.14, mentre la sua implementazione è in Figura 7.15. Un oggetto della classe **Clock** può ricevere messaggi che indicano di incrementare la

variabile membro **time** (che rappresenta l'istante corrente della simulazione) attraverso la funzione membro **tick**, di cui abbiamo il prototipo alla linea 90 e l'implementazione alle linee 111 e 112. L'istante corrente viene reso disponibile agli altri oggetti grazie alla funzione membro **getTime** presentata alle linee 91, 114 e 115. Come potete notare **getTime** è una funzione **const** poiché non ha alcun bisogno di modificare lo stato dell'oggetto sul quale è invocata.

```
80  // clock.h
81  // Definizione della classe Clock.
82  #ifndef CLOCK_H
83  #define CLOCK_H
84
85  class Clock {
86
87  public:
88        Clock(); // costruttore
89        ~Clock(); // distruttore
90        void tick(); // incrementa l'orario di un istante temporale
91              int getTime() const; // restituisce l'istante corrente
92
93  private:
94        int time; // orario
95  };
96
97  #endif // CLOCK_H
```

Figura 7.14 File di intestazione della classe Clock.

```
98  // clock.cpp
99  // Definizione delle funzioni membro della classe Clock.
100 #include <iostream.h>
101
102 #include "clock.h"
103
104 Clock::Clock() // costruttore
105       : time( 0 )
106 { cout << "clock created" << endl; }
107
108 Clock::~Clock() // distruttore
109 { cout << "clock destroyed" << endl; }
110
111 void Clock::tick() // incrementa time di 1
112 { time++; }
113
114 int Clock::getTime() const // restituisce l'istante corrente
115 { return time; }
```

Figura 7.15 File di implementazione della classe Clock.

La classe **Scheduler** (Figura 7.16) ha la responsabilità di creare gli oggetti della classe **Person** ad istanti generati in modo casuale, e di porli sui piani appropriati. L'interfaccia

public della classe comprende la funzione membro **processTime**, che riceve come argomento l'istante corrente (linea 128). Il file di intestazione contiene anche diverse funzioni di utilità **private** (le discuteremo tra un momento) che implementano alcune operazioni necessarie alla funzione membro **processTime**.

La Figura 7.17 contiene il file di implementazione della classe **Scheduler**. La funzione membro **processTime** (linee 207–217) delega la maggior parte delle sue responsabilità a funzioni di utilità più semplici della stessa classe. Il costruttore di **Scheduler** (linee 163–174) cambia innanzitutto il seme del generatore di numeri pseudocasuali con un numero che si basa sull'orario corrente del sistema (linea 168): questa operazione fa sì che il generatore di numeri casuali produca sequenze di numeri diverse ad ogni esecuzione del programma. La classe **Scheduler** chiama quindi la funzione di utilità **scheduleTime** (linee 179–192) una volta per ogni piano (linee 172-173) in modo che essa calcoli un orario di arrivo pseudocasuale (cioè un numero da 5 a 20, estremi compresi) per ciascun oggetto **Person** presente.

```
116  // scheduler.h
117  // Definizione della classe Scheduler
118  #ifndef SCHEDULER_H
119  #define SCHEDULER_H
120
121  class Floor; // dichiarazione anticipata
122
123  class Scheduler {
124
125  public:
126       Scheduler( Floor &, Floor & ); // costruttore
127       ~Scheduler(); // distruttore
128       void processTime( int ); // imposta l'orario del meccanismo
129                                // di gestione degli arrivi
130  private:
131       // programma l'arrivo al piano
132       void scheduleTime( const Floor & );
133
134       // ritarda l'arrivo al piano
135       void delayTime( const Floor & );
136
137       // crea una nuova persona e la pone su un piano
138       void createNewPerson( Floor & );
139
140       // gestisce l'arrivo della persona al piano
141       void handleArrivals( Floor &, int );
142
143       int currentClockTime;
144
145       Floor &floor1Ref;
146       Floor &floor2Ref;
```

Figura 7.16 File di intestazione della classe Scheduler (continua)


```
147
148       int floor1ArrivalTime;
149       int floor2ArrivalTime;
150  };
151
152  #endif // SCHEDULER_H
```

Figura 7.16 File di intestazione della classe Scheduler

```
153  // scheduler.cpp
154  // Definizione delle funzioni membro della classe Scheduler.
155  #include <iostream.h
156  #include <stdlib.h>
157  #include <time.h>
158
159  #include "scheduler.h"
160  #include "floor.h"
161  #include "person.h"
162
163  // costruttore
164  Scheduler::Scheduler( Floor &firstFloor, Floor &secondFloor )
165       : currentClockTime( 0 ), floor1Ref( firstFloor ),
166       floor2Ref( secondFloor )
167  {
168       srand( time( 0 ) ); //nuovo seme per il gen. di numeri cas.
169       cout << "scheduler created" << endl;
170
171       // programma i primi arrivi sui due piani
172       scheduleTime( floor1Ref );
173       scheduleTime( floor2Ref );
174  }
175
176  Scheduler::~Scheduler() // distruttore
177  { cout << "scheduler destroyed" << endl; }
178
179  // programma l'arrivo su un piano
180  void Scheduler::scheduleTime( const Floor &floor )
181  {
182       int floorNumber = floor.getNumber();
183       int arrivalTime = currentClockTime + ( 5 + rand() % 16 );
184
185       floorNumber == Floor::FLOOR1 ?
186       floor1ArrivalTime = arrivalTime :
187       floor2ArrivalTime = arrivalTime;
188
189       cout << "(scheduler schedules next person for floor "
190            << floorNumber << " at time " << arrivalTime << ')'
191            << endl;
192  }
```

Figura 7.17 File di implementazione della classe Scheduler (continua)

```
193
194  // riprogramma un arrivo
195  void Scheduler::delayTime( const Floor &floor )
196  {
197       int floorNumber = floor.getNumber();
198
199       int arrivalTime = ( floorNumber == Floor::FLOOR1 ) ?
200       ++floor1ArrivalTime : ++floor2ArrivalTime;
201
202       cout << "(scheduler delays next person for floor "
203            << floorNumber << " until time " << arrivalTime << ')'
204            << endl;
205  }
206
207  // passa l'orario al meccanismo di gestione degli arrivi
208  void Scheduler::processTime( int time )
209  {
210       currentClockTime = time; // annota l'orario
211
212       // gestisce gli arrivi al primo piano
213       handleArrivals( floor1Ref, currentClockTime );
214
215       // gestisce gli arrivi al secondo piano
216       handleArrivals( floor2Ref, currentClockTime );
217  }
218
219  // crea una nuova persona e la pone su un piano specificato
220  void Scheduler::createNewPerson( Floor &floor )
221  {
222       int destinationFloor =
223       floor.getNumber() == Floor::FLOOR1 ?
224       Floor::FLOOR2 : Floor::FLOOR1;
225
226  // crea una nuova persona
227       Person *newPersonPtr = new Person( destinationFloor );
228
229       cout << "scheduler creates person "
230            << newPersonPtr->getID() << endl;
231
232       // pone la persona sul piano appropriato
233       newPersonPtr->stepOntoFloor( floor );
234
235       scheduleTime( floor ); // programma l'arrivo successivo
236  }
237
238  // gestisce gli arrivi su un piano specificato
239  void Scheduler::handleArrivals( Floor &floor, int time )
240  {
```

Figura 7.17 File di implementazione della classe Scheduler (continua)


```
241       int floorNumber = floor.getNumber();
242
243       int arrivalTime = ( floorNumber == Floor::FLOOR1 ) ?
244       floor1ArrivalTime : floor2ArrivalTime;
245
246       if ( arrivalTime == time )
247       {
248            if ( floor.isOccupied() ) //controlla: piano occupato?
249                 delayTime( floor );
250            else
251                 createNewPerson( floor );
252       }
253  }
```

Figura 7.17 File di implementazione della classe Scheduler.

Nella nostra simulazione **building** passa al trascorrere di ogni istante l'orario aggiornato a **scheduler** tramite la funzione membro **processTime** di quest'ultimo(linee 207–217). Il diagramma di sequenza in Figura 4.27 rappresentava la sequenza delle attività che si verificano in risposta a tale messaggio, e la nostra implementazione riflette appunto questo modello. Quando viene invocata la funzione membro **processTime**, **scheduler** chiama la funzione di utilità **handleArrivals** per i due piani (linee 213 e 216): essa confronta il valore corrente di **time** fornito da **building** con l'istante di arrivo programmato per il piano appropriato (linea 246). Se i due valori sono uguali e se il piano è correntemente occupato (linea 248), **scheduler** chiama la funzione di utilità **delayTime** per ritardare di un istante l'arrivo successivo programmato (linea 249). Se il piano è libero, **scheduler** invoca la funzione di utilità **createNewPerson** (linea 251), che crea un nuovo oggetto della classe **Person** con l'operatore **new** (linea 227). L'oggetto **scheduler** invia quindi il messaggio **stepOntoFloor** al nuovo oggetto della classe **Person** (linea 233). Non appena la persona accede al piano, **scheduler** calcola l'orario di arrivo della persona successiva per quel piano chiamando la funzione di utilità **scheduleTime** (linea 235).

Abbiamo analizzato l'implementazione di tutte le classi che compongono la parte di controllo della simulazione, per cui possiamo passare adesso ad esaminarne la parte che rappresenterà il mondo reale. La classe **Bell**, come **Clock**, non è composta da altri oggetti e la sua interfaccia pubblica, definita nel file di intestazione in Figura 7.18, contiene un costruttore, un distruttore e la funzione membro **ringBell**. L'implementazione di queste funzioni (Figura 7.19, rispettivamente linee 274–275, 277–278 e 280–281) invia semplicemente messaggi in output sullo schermo, e nel caso di **ringBell** utilizza la sequenza di escape **'\a'** che ha come effetto quello di emettere un beep.

```
254  // bell.h
255  // Definizione della classe Bell.
256  #ifndef BELL_H
257  #define BELL_H
258
259  class Bell {
260
261  public:
```

Figura 7.18 File di intestazione della classe Bell (continua)

```
262       Bell(); // costruttore
263       ~Bell(); // distruttore
264       void ringBell() const; // suona il campanello
265  };
266
267  #endif // BELL_H
```

Figura 7.18 File di intestazione della classe Bell.

```
268  // bell.cpp
269  // Definizioni delle funzioni membro della classe Bell.
270  #include <iostream.h>
271
272  #include "bell.h"
273
274  Bell::Bell() // costruttore
275  { cout << "bell created" << endl; }
276
277  Bell::~Bell() // distruttore
278  { cout << "bell destroyed" << endl; }
279
280  void Bell::ringBell() const // suona il campanello
281  { cout << "elevator rings its bell\a" << endl; }
```

Figura 7.19 File di implementazione della classe Bell.

La classe `Light` (Figura 7.20 e 7.21) ha un'interfaccia pubblica costituita da due funzioni membro, un costruttore e un distruttore. La funzione membro `turnOn` accende semplicemente la spia impostando il dato membro `on` su `true` (linee 314–318). La funzione membro `turnOff` (linee 320–324) la spegne impostando il dato membro `on` su `false`.

```
282  // light.h
283  // Definizione della classe Light.
284  #ifndef LIGHT_H
285  #define LIGHT_H
286
287  class Light {
288
289  public:
290       Light( const char * ); // costruttore
291       ~Light(); // distruttore
292       void turnOn(); // accende la spia
293       void turnOff(); // spegne la spia
294
295  private:
296       bool on; // true se accesa; false se spenta
297       const char *name; // piano in cui si trova la spia
298  };
299
300  #endif // LIGHT_H
```

Figura 7.20 File di intestazione della classe Light.


```
301  // light.cpp
302  // Definizioni delle funzioni membro della classe Light.
303  #include <iostream.h>
304
305  #include "light.h"
306
307  Light::Light( const char *string ) // costruttore
308       : on( false ), name( string )
309  { cout << name << " light created" << endl; }
310
311  Light::~Light() // distruttore
312  { cout << name << " light destroyed" << endl; }
313
314  void Light::turnOn() // accende la spia
315  {
316       on = true;
317       cout << name << " turns on its light" << endl;
318  }
319
320  void Light::turnOff() // spegne la spia
321  {
322       on = false;
323       cout << name << " turns off its light" << endl;
324  }
```

Figura 7.21 File di implementazione della classe Light.

La classe `Door` (Figura 7.22 e 7.23) ha un ruolo importante nella simulazione. È infatti l'oggetto `door` che segnala al passeggero di uscire o alla persona in attesa al piano di entrare nell'ascensore attraverso la funzione membro `openDoor` della classe. Avrete notato che `openDoor` prende quattro argomenti (linee 340-341 e 366-368): il primo è un puntatore all'oggetto della classe `Person` che occupa l'ascensore, il secondo è un puntatore all'oggetto della classe `Person` in attesa al piano, mentre i restanti due argomenti sono riferimenti agli oggetti appropriati della classe `Floor` ed `elevator`.

```
325  // door.h
326  // Definizione della classe Door.
327  #ifndef DOOR_H
328  #define DOOR_H
329
330  class Person; // dichiarazione anticipata
331  class Floor; // dichiarazione anticipata
332  class Elevator; // dichiarazione anticipata
333
334  class Door {
335
336  public:
337       Door(); // costruttore
338       ~Door(); // distruttore
```

Figura 7.22 File di intestazione della classe Door (continua)

```
339
340       void openDoor( Person * const, Person * const,
341            Floor &, Elevator & );
342       void closeDoor( const Floor & );
343
344   private:
345       bool open; // aperta o chiusa
346   };
347
348   #endif // DOOR_H
```

Figura 7.22 File di intestazione della classe Door.

La classe `Door` è un oggetto componente della classe `Elevator`: per implementare questa relazione di composizione, includiamo nel file di intestazione della classe `Elevator` la linea

```
#include "door.h"
```

La classe `Door` utilizza un riferimento a un oggetto della classe `Elevator` (linea 341). Per dichiarare la classe `Elevator` in modo tale che la classe `Door` possa utilizzare questo riferimento, potremmo scrivere questa linea nel file di intestazione della classe `Door`:

```
#include "elevator.h"
```

In questo modo però il file di intestazione della classe `Elevator` includerebbe quello della classe `Door` e viceversa ed il preprocessore non sarebbe in grado di risolvere queste direttive poiché si richiamano circolarmente l'una con l'altra.

Per evitare questo problema scriviamo una dichiarazione anticipata della classe `Elevator` nel file di intestazione della classe `Door` (linea 332), che indica al preprocessore che nel file ci sono riferimenti a oggetti della classe `Elevator`, ma che la definizione di tale classe si trova in un altro file. Come potete notare abbiamo incluso anche dichiarazioni anticipate delle classi `Person` e `Floor` (linee 330–331), per cui possiamo far riferimento a loro nel prototipo della funzione membro `openDoor`.

La Figura 7.23 contiene il file di implementazione della classe `Door`. Alle linee 354-356 includiamo i file di intestazione delle classi `Person`, `Floor` ed `Elevator`: queste direttive `#include` corrispondono alle nostre dichiarazioni anticipate nel file di intestazione, e i file di intestazione che vengono inclusi contengono i prototipi di funzione obbligatori che consentono di invocare nel modo appropriato le funzioni membro di queste classi.

Quando viene chiamata la funzione membro `openDoor` (linee 365–385), controlla innanzitutto che la porta non sia già aperta.

L'oggetto `door` controlla che il puntatore alla persona nell'ascensore (`passengerPtr`) non sia uguale a zero (linea 376). In tal caso, significa che nell'ascensore c'è una persona che deve uscire, e che viene quindi "invitata" ad uscire tramite il messaggio `exitElevator` (linea 377). A questo punto l'oggetto `door` elimina tale oggetto `Person` con l'operatore `delete` (linea 378).


```
349  // door.cpp
350  // Definizioni delle funzioni membro della classe Door.
351  #include <iostream.h>
352
353  #include "door.h"
354  #include "person.h"
355  #include "floor.h"
356  #include "elevator.h"
357
358  Door::Door() // costruttore
359       : open( false )
360  { cout << "door created" << endl; }
361
362  Door::~Door() // distruttore
363  { cout << "door destroyed" << endl; }
364
365  // apre la porta
366  void Door::openDoor( Person * const passengerPtr,
367       Person * const nextPassengerPtr,
368       Floor &currentFloor, Elevator &elevator )
369  {
370       if ( !open ) {
371            open = true;
372
373            cout << "elevator opens its door on floor "
374                 << currentFloor.getNumber() << endl;
375
376            if ( passengerPtr != 0 ) {
377              passengerPtr->exitElevator( currentFloor,Elevator );
378              delete passengerPtr;//il pass. abbandona simulazione
379            }
380
381            if ( nextPassengerPtr != 0 )
382                 nextPassengerPtr->enterElevator(
383                      elevator, currentFloor );
384       }
385  }
386
387  // chiude la porta
388  void Door::closeDoor( const Floor &currentFloor )
389  {
390       if ( open ) {
391            open = false;
392            cout << "elevator closes its door on floor "
393                 << currentFloor.getNumber() << endl;
394       }
395  }
```

Figura 7.23 File di implementazione della classe Door.

Quando il passeggero esce dall'ascensore, la porta controlla che il puntatore alla persona in attesa al piano (`nextPassengerPtr`) non sia nullo (linea 381). In tal caso significa *che c'è una persona che attende di entrare nell'ascensore, e può farlo tramite la funzione* membro `enterElevator` della classe `Person` (linee 382-383). La funzione membro `closeDoor` della classe `Door` (linee 388–395) controlla semplicemente se la porta è aperta e, se è così, la chiude.

Le persone del sistema utilizzano un oggetto della classe `ElevatorButton` (Figura 7.24 e 7.25) per ordinare ad `elevator` di cominciare la corsa verso l'altro piano. La funzione membro `pressButton` (linee 433-439) imposta innanzitutto l'attributo `pressed` del pulsante interno su `true` e invia il messaggio `prepareToLeave` ad `elevator`. La funzione membro `resetButton` imposta semplicemente l'attributo `pressed` su `false` (cioè rilascia il pulsante).

```
396  // elevatorButton.h
397  // Definizione della classe ElevatorButton.
398  #ifndef ELEVATORBUTTON_H
399  #define ELEVATORBUTTON_H
400
401  class Elevator; // dichiarazione anticipata
402
403  class ElevatorButton {
404
405  public:
406       ElevatorButton( Elevator & ); // costruttore
407       ~ElevatorButton(); // distruttore
408
409       void pressButton(); // preme il pulsante
410       void resetButton(); // rilascia il pulsante
411
412  private:
413       bool pressed; // stato del pulsante
414       Elevator &elevatorRef; // rif. del pulsante all'ascensore
415  };
416
417  #endif // ELEVATORBUTTON_H
```

Figura 7.24 File di intestazione della classe ElevatorButton.

```
418  // elevatorButton.cpp:
419  // Definizioni delle funzioni membro della classe ElevatorButton.
420  #include <iostream.h>
421
422  #include "elevatorButton.h"
423  #include "elevator.h"
424
425  // costruttore
426  ElevatorButton::ElevatorButton( Elevator &elevatorHandle )
427       : pressed( false ), elevatorRef( elevatorHandle )
```

Figura 7.25 File di implementazione della classe ElevatorButton (continua)


```
428  { cout << "elevator button created" << endl; }
429
430  ElevatorButton::~ElevatorButton() // distruttore
431  { cout << "elevator button destroyed" << endl; }
432
433  void ElevatorButton::pressButton() // preme il pulsante
434  {
435       pressed = true;
436       cout <<"elevator button tells elevator to prepare to leave"
437            << endl;
438       elevatorRef.prepareToLeave( true );
439  }
440
441  void ElevatorButton::resetButton() // rilascia il pulsante
442  { pressed = false; }
```

Figura 7.25 File di implementazione della classe ElevatorButton.

L'interfaccia pubblica della classe `FloorButton` (Figura 7.26 e 7.27) contiene le stesse funzioni membro della classe `ElevatorButton`. La funzione membro `pressButton`, di tipo `public`, chiama `elevator` tramite il messaggio `summonElevator`. Il pulsante esterno viene rilasciato con una chiamata alla funzione membro `resetButton`.

```
443  // floorButton.h
444  // Definizione della classe FloorButton.
445  #ifndef FLOORBUTTON_H
446  #define FLOORBUTTON_H
447
448  class Elevator; // dichiarazione anticipata
449
450  class FloorButton {
451
452  public:
453       FloorButton( const int, Elevator & ); // costruttore
454       ~FloorButton(); // distruttore
455
456       void pressButton(); // preme il pulsante
457       void resetButton(); // rilascia il pulsante
458
459  private:
460       const int floorNumber; // n° del piano relativo al pulsante
461       bool pressed; // stato del pulsante
462
463       // riferimento del pulsante all'ascensore
464       Elevator &elevatorRef;
465  };
466
467  #endif // FLOORBUTTON_H
```

Figura 7.26 File di intestazione della classe FloorButton.

```
468  // floorButton.cpp
469  // Definizione delle funzioni membro della classe FloorButton.
470  #include <iostream.h>
471
472  #include "floorButton.h"
473  #include "elevator.h"
474
475  // costruttore
476  FloorButton::FloorButton( const int number,
477       Elevator &elevatorHandle )
478       :floorNumber( number ), pressed( false ),
479       elevatorRef( elevatorHandle )
480  {
481       cout << "floor " << floorNumber << " button created"
482            << endl;
483  }
484
485  FloorButton::~FloorButton() // distruttore
486  {
487       cout << "floor " << floorNumber << " button destroyed"
488            << endl;
489  }
490
491  // preme il pulsante
492  void FloorButton::pressButton()
493  {
494       pressed = true;
495       cout << "floor " << floorNumber
496            << " button summons elevator" << endl;
497       elevatorRef.summonElevator( floorNumber );
498  }
499
500  // rilascia il pulsante
501  void FloorButton::resetButton()
502  { pressed = false; }
```

Figura 7.27 File di implementazione della classe FloorButton.

Il file di intestazione della classe **Elevator** (Figura 7.28) è il più complesso della nostra simulazione. La classe **Elevator** include cinque funzioni membro pubbliche, oltre al costruttore e al distruttore. La funzione membro **processTime** consente all'edificio di inviare all'ascensore il valore aggiornato di **time**. La funzione membro **summonElevator** è utilizzata dai pulsanti **FloorButton**per per richiedere il servizio all'ascensore. Le funzioni membro **passengerEnters** e **passengerExits** consentono ai passeggeri di entrare e uscire dall'ascensore, mentre la funzione membro **prepareToLeave** è utilizzata dall'ascensore per effettuare le operazioni preliminari necessarie prima di iniziare una corsa. Dichiariamo **public** l'oggetto **elevatorButton** in modo tale che un oggetto della classe **Person** possa accedervi direttamente.

Le funzioni di utilità si trovano alle linee 528–531. La classe **Elevator** definisce anche una serie di costanti di classe private (linee 534–536) che contengono informazioni utilizzate contemporaneamente da tutti gli oggetti della classe **Elevator**. Inoltre queste informazioni non devono essere modificati, e questo è il motivo per cui sono dichiarate come costanti.

```
503  // elevator.h
504  // Definizione della classe Elevator.
505  #ifndef ELEVATOR_H
506  #define ELEVATOR_H
507
508  #include "elevatorButton.h"
509  #include "door.h"
510  #include "bell.h"
511
512  class Floor; // dichiarazione anticipata
513  class Person; // dichiarazione anticipata
514
515  class Elevator {
516
517  public:
518       Elevator( Floor &, Floor & ); // costruttore
519       ~Elevator(); // distruttore
520       void summonElevator( int ); //richiesta di serv. a un piano
521       void prepareToLeave( bool ); // preparazione alla corsa
522       void processTime( int ); // passa l'orario all'ascensore
523       void passengerEnters( Person * const ); // il pass. entra
524       void passengerExits(); // il passeggero esce
525       ElevatorButton elevatorButton; // notare l'oggetto public
526
527  private:
528       void processPossibleArrival();
529       void processPossibleDeparture();
530       void arriveAtFloor( Floor & );
531       void move();
532
533       // tempo di percorrenza
534       static const int ELEVATOR_TRAVEL_TIME;
535       static const int UP; // direzione verso l'alto
536       static const int DOWN; // direzione verso il basso
537
538       int currentBuildingClockTime; // orario corrente
539       bool moving; // stato dell'ascensore
540       int direction; // direzione corrente
541       int currentFloor; // locazione corrente
542       int arrivalTime; // orario di arrivo al piano
543       bool floor1NeedsService; // flag di serv. del primo piano
544       bool floor2NeedsService; // flag di servizio del sec. piano
```

Figura 7.28 File di intestazione della classe Elevator (continua)

```
545
546       Floor &floor1Ref; // riferimento al primo piano
547       Floor &floor2Ref; // riferimento al secondo piano
548       Person *passengerPtr; // puntatore al passeggero corrente
549
550       Door door; // oggetto "porta"
551       Bell bell; // oggetto "campanello"
552    };
553
554    #endif // ELEVATOR_H
```

Figura 7.28 File di intestazione della classe Elevator.

Le linee 538–551 del file di intestazione di `Elevator` contengono altri dati membro `private`. Notate gli handle nella forma di riferimenti a tutti gli oggetti della classe `Floor` (linee 546–547), mentre nella forma di puntatore per l'oggetto "passeggero" (linea 548). Questa scelta si spiega con il fatto che mentre gli oggetti della classe `Floor` esisteranno e *non verranno modificati per tutta la durata della simulazione*, il *passeggero*, *invece*, cambierà ogni volta che un oggetto `Person` entra o esce dall'ascensore.

Nelle figure 3.31, 3.32 e 5.37 abbiamo utilizzato UML per rappresentare diverse attività e collaborazioni associate alla classe `Elevator`, ed il codice della classe (Figura 7.29) implementa tutte le specifiche presenti in quei diagrammi. Il costruttore di `Elevator` ha un elenco di inizializzatori di membro piuttosto cospicuo (linee 569–574). Ricorderete dalla definizione di `ElevatorButton` (Figura 7.24) che tale classe deve disporre di un handle a un oggetto della classe `Elevator`, che le viene passato come argomento del costruttore. Tale handle viene fornito nell'elenco degli inizializzatori di membro, dereferenziando il puntatore `this` di `elevator` (linea 569). Alcuni compilatori potrebbero generare un avvertimento (o warning) in corrispondenza di questa istruzione perché a questo punto `elevator` non è stato ancora inizializzato completamente.

```
555  // elevator.cpp
556  // Definizioni delle funzioni membro della classe Elevator.
557  #include <iostream.h>
558
559  #include "elevator.h"
560  #include "person.h"
561  #include "floor.h"
562
563  const int Elevator::ELEVATOR_TRAVEL_TIME = 5;
564  const int Elevator::UP = 0;
565  const int Elevator::DOWN = 1;
566
567  // costruttore
568  Elevator::Elevator( Floor &firstFloor, Floor &secondFloor )
569     : elevatorButton( *this ), currentBuildingClockTime( 0 ),
570     moving( false ), direction( UP ),
571     currentFloor( Floor::FLOOR1 ), arrivalTime( 0 ),
572     floor1NeedsService( false ), floor2NeedsService( false ),
```

Figura 7.29 File di implementazione della classe Elevator (continua)


```
573       floor1Ref( firstFloor ), floor2Ref( secondFloor ),
574       passengerPtr( 0 )
575  { cout << "elevator created" << endl; }
576
577  Elevator::~Elevator() // distruttore
578  { cout << "elevator destroyed" << endl; }
579
580  // passa l'orario all'ascensore
581  void Elevator::processTime( int time )
582  {
583       currentBuildingClockTime = time;
584
585       if ( moving )
586            processPossibleArrival();
587       else
588            processPossibleDeparture();
589
590       if ( !moving )
591            cout << "elevator at rest on floor "
592                 << currentFloor << endl;
593  }
594
595  // quando l'ascensore è in moto, determina se deve fermarsi
596  void Elevator::processPossibleArrival()
597  {
598       // se l'ascensore arriva al piano di destinazione
599       if ( currentBuildingClockTime == arrivalTime ) {
600
601            currentFloor = // update current floor
602                 ( currentFloor == Floor::FLOOR1 ?
603                 Floor::FLOOR2 : Floor::FLOOR1 );
604
605            direction = // aggiorna la direzione
606                 ( currentFloor == Floor::FLOOR1 ? UP : DOWN );
607
608            cout << "elevator arrives on floor "
609                 << currentFloor << endl;
610
611            arriveAtFloor( currentFloor == Floor::FLOOR1 ?
612                 floor1Ref : floor2Ref );
613
614            return;
615       }
616
617       // l'ascensore è in corsa
618       cout << "elevator moving "
619            << ( direction == UP ? "up" : "down" ) << endl;
620  }
```

Figura 7.29 File di implementazione della classe Elevator (continua)

```
621
622  // determina se l'ascensore deve spostarsi
623  void Elevator::processPossibleDeparture()
624  {
625       // deve essere servito questo piano?
626       bool currentFloorNeedsService =
627       currentFloor == Floor::FLOOR1 ?
628            floor1NeedsService : floor2NeedsService;
629
630       // deve essere servito l'altro piano?
631       bool otherFloorNeedsService =
632       currentFloor == Floor::FLOOR1 ?
633            floor2NeedsService : floor1NeedsService;
634
635       // serve questo piano (se necessario)
636       if ( currentFloorNeedsService ) {
637            arriveAtFloor( currentFloor == Floor::FLOOR1 ?
638                 floor1Ref : floor2Ref );
639
640            return;
641       }
642
643       // serve l'altro piano (se necessario)
644       else prepareToLeave( otherFloorNeedsService );
645  }
646
647  // arriva ad un piano particolare
648  void Elevator::arriveAtFloor( Floor& arrivalFloor )
649  {
650       moving = false; // aggiorna lo stato
651
652       cout << "elevator resets its button" << endl;
653       elevatorButton.resetButton();
654
655       bell.ringBell();
656
657       // notifica al piano l'arrivo dell'ascensore
658       Person *floorPersonPtr = arrivalFloor.elevatorArrived();
659
660       door.openDoor( passengerPtr, floorPersonPtr,
661            arrivalFloor, *this );
662
663       // deve essere servito questo piano?
664       bool currentFloorNeedsService =
665       currentFloor == Floor::FLOOR1 ?
666            floor1NeedsService : floor2NeedsService;
667
```

Figura 7.29 File di implementazione della classe Elevator (continua)


```
668       // deve essere servito l'altro piano?
669       bool otherFloorNeedsService =
670       currentFloor == Floor::FLOOR1 ?
671            floor2NeedsService : floor1NeedsService;
672
673       // se non deve essere servito questo piano
674       // l'ascensore deve prepararsi ad andare all'altro piano
675       if ( !currentFloorNeedsService )
676            prepareToLeave( otherFloorNeedsService );
677       else // altrimenti, disabilita il flag di servizio
678         currentFloor == Floor::FLOOR1 ?
679           floor1NeedsService = false: floor2NeedsService = false;
680  }
681
682  // richiesta di servizio
683  void Elevator::summonElevator( int floor )
684  {
685       // imposta il flag di servizio appropriato
686       floor == Floor::FLOOR1 ?
687            floor1NeedsService = true : floor2NeedsService = true;
688  }
689
690  // accetta un passeggero
691  void Elevator::passengerEnters( Person * const personPtr )
692  {
693       // il passeggero entra
694       passengerPtr = personPtr;
695
696       cout << "person " << passengerPtr->getID()
697            << " enters elevator from floor "
698            << currentFloor << endl;
699  }
700
701  // notifica all'ascensore l'uscita del passeggero
702  void Elevator::passengerExits() { passengerPtr = 0; }
703
704  // preparazione ad abbandonare il piano
705  void Elevator::prepareToLeave( bool leaving )
706  {
707       Floor &thisFloor =
708       currentFloor == Floor::FLOOR1 ? floor1Ref : floor2Ref;
709
710  // notifica al piano che l'ascensore potrebbe spostarsi a breve
711       thisFloor.elevatorLeaving();
712
713       door.closeDoor( thisFloor,
714
```

Figura 7.29 File di implementazione della classe Elevator (continua)

```
715       if ( leaving ) // corsa (se necessario)
716          move();
717    }
718
719    void Elevator::move() // vai al piano determinato
720    {
721       moving = true; // cambia stato
722
723       // programma l'orario di arrivo
724       arrivalTime = currentBuildingClockTime +
725          ELEVATOR_TRAVEL_TIME;
726
727       cout << "elevator begins moving "
728          << ( direction == DOWN ? "down " : "up " )
729          << "to floor "
730          << ( direction == DOWN ? '1' : '2' )
731          << " (arrives at time " << arrivalTime << ')'
732          << endl;
733    }
```

Figura 7.29 File di implementazione della classe Elevator.

L'oggetto `building` invoca la funzione membro `processTime` (linee 580–593) della classe `Elevator`, passandole come parametro il valore corrente di `time`. Questa funzione membro aggiorna il dato membro `currentBuildingClockTime` assegnandogli il valore corrente di `time` (linea 583) e controlla quindi il valore del dato membro `motion` (linea 585). Se l'ascensore è in corsa, esso invoca la propria funzione di utilità `processPossibleArrival` (linea 586), altrimenti invoca `processPossibleDeparture` (linea 588). In questo modo l'ascensore determina se deve arrivare al piano corrente o se deve lasciarlo per andare all'altro piano, mentre se non deve spostarsi visualizza un messaggio sullo schermo che indica che è fermo al piano `currentFloor` (linee 590–592).

La funzione `processPossibleArrival` determina se l'ascensore debba fermarsi confrontando il valore di `currentBuildingClockTime` con il valore di arrivo al piano calcolato (linea 599). Se l'istante corrente corrisponde con quello di arrivo al piano di destinazione, `elevator` aggiorna il piano corrente `currentFloor` (linee 601–603) e la direzione della corsa `direction` (linee 605–606), quindi chiama la propria funzione di utilità `arriveAtFloor` per portare a termine le operazioni necessarie a una fermata al piano.

La funzione di utilità `processPossibleDeparture` determina se l'ascensore debba cominciare una corsa verso l'altro piano determinando quale sia il piano che richiede il servizio (linee 625–633). Nel caso sia il piano corrente, `elevator` chiama la propria funzione di utilità `arriveAtFloor` per il piano corrente (linee 637–648), altrimenti chiama `prepareToLeave` (linea 644) e si sposta verso l'altro piano.

La funzione di utilità `arriveAtFloor` effettua le operazioni necessarie perché `elevator` arrivi ad un piano, cioè: ferma l'ascensore impostando il membro `moving` su `false` (linea 650), rilascia il pulsante interno `elevatorButton` (linea 653) e suona il campanello `bell` (linea 655). A quel punto viene dichiarato un puntatore temporaneo a un oggetto della

classe `Person`, destinato a memorizzarvi un eventuale handle a un oggetto `Person` in attesa al piano (potrebbe darsi che nessuna persona stia attendendo in quel momento). Questo puntatore riceve il valore restituito dalla funzione `elevatorArrived` del piano (linea 658).

L'ascensore apre la porta invocando la funzione membro `openDoor` della classe `Door` e passandole come parametri un handle al passeggero corrente, un handle alla persona in attesa al piano, un handle al piano dove è arrivato l'ascensore e un handle allo stesso `elevator` (linee 660–661). A quel punto `elevator` determina nuovamente se uno dei due piani richiede il servizio (linee 663–671). Se non è così, esso aggiorna il flag di servizio relativo al piano corrente (linee 678–679). Altrimenti se il servizio non è richiesto dal piano corrente, `elevator` si prepara a partire per l'altro piano (linea 676) e lo fa se l'altro piano lo richiede effettivamente..

La funzione membro `summonElevator` consente il servizio di `elevator` da parte di altri oggetti. Essa riceve come argomento il numero del piano ed imposta il flag di servizio appropriato su `true` (linee 686 e 687).

La funzione membro `passengerEnters` ha come unico argomento un puntatore a un oggetto della classe `Person` (linea 691), e aggiorna l'handle `passengerPtr` di `elevator` in modo che punti al nuovo passeggero (linea 694). La funzione membro `passengerExits` imposta semplicemente l'handle `passengerPtr` a zero, indicando così che non è presente alcun passeggero nell'ascensore (linea 702).

La funzione membro `prepareToLeave` riceve un argomento di tipo `bool` che indica se l'ascensore deve spostarsi all'altro piano (linea 705). L'ascensore notifica il suo spostamento al piano corrente inviandogli il messaggio `elevatorLeaving` (linea 711), quindi chiude quindi la porta `door` (linea 713) edinfine controlla se ha bisogno di lasciare il piano (linea 715). Se è così, esso comincia la corsa chiamando la funzione di utilità `move` (linea 716), che imposta il dato membro `moving` su `true` (linea 721). Il codice calcola poi l'orario di arrivo al piano di destinazione utilizzando la costante di classe`ELEVATOR_TRAVEL_TIME` (linee 724–725). Infine l'ascensore visualizza la direzione (`direction`) della corsa, il piano di destinazione e l'orario di arrivo programmato `arrivalTime` (linee 727–732).

La definizione della classe `Floor` (Figura 7.30) associa gli oggetti di altre classi agli oggetti `Floor` in diversi modi. Prima definisce un riferimento come handle ad `elevator` (linea 771), che è una soluzione appropriata, in quanto questo handle si riferisce sempre allo stesso ascensore. Troviamo poi un puntatore come handle a un oggetto `Person` (linea 772): questo handle cambierà ogni volta che arriva o se ne va una nuova persona dalla simulazione. Infine abbiamo degli oggetti composti, tra cui un oggetto `floorButton` di visibilità `public` (linea 767) e un oggetto `light` di visibilità `private` (linea 773). Dichiariamo `floorButton` come `public` per consentire agli oggetti `Person` di accedervi direttamente. La definizione della classe `Floor` contiene anche le costanti di classe di nome `FLOOR1` e `FLOOR2` (linee 765-766), due costanti che utilizzeremo al posto del numero dei piani e che sono inizializzate nel file di implementazione (linee 785-786). Normalmente i dati membro di tipo `const` di una classe devono essere inizializzati nell'elenco degli inizializzatori di membro del costruttore, ma nel caso particolare di dati membro `static const`, essi sono inizializzati con al livello di visibilità di file.

```
734  // floor.h
735  // Definizione della classe Floor.
736  #ifndef FLOOR_H
737  #define FLOOR_H
738
739  #include "floorButton.h"
740  #include "light.h"
741
742  class Elevator; // dichiarazione anticipata
743  class Person; // dichiarazione anticipata
744
745  class Floor {
746
747  public:
748       Floor( int, Elevator & ); // costruttore
749       ~Floor(); // distruttore
750       bool isOccupied() const; //restituisce true se piano occ.to
751       int getNumber() const; // restituisce il numero del piano
752
753       // passa un handle alla nuova persona al piano
754       void personArrives( Person * const );
755
756       // notifica al piano l'arrivo dell'ascensore
757       Person *elevatorArrived();
758
759       // notifica al piano la partenza dell'ascensore
760       void elevatorLeaving();
761
762       // comunica al piano che la persona sta lasciando il piano
763       void personBoardingElevator();
764
765       static const int FLOOR1;
766       static const int FLOOR2;
767       FloorButton floorButton; // oggetto "pulsante esterno"
768
769  private:
770       const int floorNumber; // numero del piano
771       Elevator &elevatorRef; // puntatore all'ascensore
772       Person *occupantPtr; // puntatore alla persona al piano
773       Light light; // oggetto "spia"
774  };
775
776  #endif // FLOOR_H
```

Figura 7.30 File di intestazione della classe Floor.

La Figura 7.31 mostra il file di implementazione della classe `Floor`. La funzione membro `isOccupied` (linee 800–802) restituisce un valore di tipo `bool` che indica se c'è una persona in attesa al piano; essa effettua questa operazione verificando che il valore di `occupantPtr` sia diverso da zero (linea 802). La funzione membro `getNumber` restituisce

il valore della variabile `floorNumber` (linea 805). La funzione membro `personArrives` riceve un puntatore all'oggetto `Person` che accede al piano e lo assegna al dato membro privato `occupantPtr`.

```
777  // floor.cpp
778  // Definizione delle funzioni membro della classe Floor.
779  #include <iostream.h>
780
781  #include "floor.h"
782  #include "person.h"
783  #include "elevator.h"
784
785  const int Floor::FLOOR1 = 1;
786  const int Floor::FLOOR2 = 2;
787
788  // costruttore
789  Floor::Floor(int number, Elevator &elevatorHandle )
790       : floorButton( number, elevatorHandle ),
791       floorNumber( number ), elevatorRef( elevatorHandle ),
792       occupantPtr ( 0 ),
793       light( floorNumber == 1 ? "floor 1" : "floor 2" )
794  { cout << "floor " << floorNumber << " created" << endl; }
795
796  // distruttore
797  Floor::~Floor()
798  { cout << "floor " << floorNumber << " destroyed" << endl; }
799
800  // determina se il piano è occupato
801  bool Floor::isOccupied() const
802  { return ( occupantPtr != 0 ); }
803
804  // restituisce il numero di questo piano
805  int Floor::getNumber() const { return floorNumber; }
806
807  // passa la "persona" al piano
808  void Floor::personArrives( Person * const personPtr )
809  { occupantPtr = personPtr; }
810
811  // notifica al piano l'arrivo dell'ascensore
812  Person *Floor::elevatorArrived()
813  {
814       // rilascia il pulsante esterno, se necessario
815       cout << "floor " << floorNumber
816            << " resets its button" << endl;
817       floorButton.resetButton();
818
819       light.turnOn();
820
```

Figura 7.31 File di implementazione della classe Floor (continua)

```
821         return occupantPtr;
822   }
823
824   // comunica al piano che l'ascensore sta partendo
825   void Floor::elevatorLeaving() { light.turnOff(); }
826
827   // comunica al piano che una persona lo sta abbandonando
828   void Floor::personBoardingElevator() { occupantPtr = 0; }
```

Figura 7.31 File di implementazione della classe Floor.

La funzione membro `elevatorArrived` (linee 811–822) rilascia l'oggetto `floorButton` del piano corrente (linea 817), accende la spia (`light`) e restituisce l'handle al passeggero contenuto nel dato membro `occupantPtr` (linea 821). La funzione membro `elevatorLeaving` spegne la spia (linea 825), infine la funzione membro `personBoardingElevator` assegna il valore nullo a `occupantPtr`, ad indicare che la persona ha lasciato il piano (linea 828).

Gli elementi del file di intestazione della classe `Person` (Figura 7.32) a questo punto dovrebbero esservi familiari. La funzione membro `getID` restituisce l'`ID` univoco di un oggetto `Person`. Le funzioni membro `stepOntoFloor`, `enterElevator` ed `exitElevator` compongono il resto dell'interfaccia pubblica della classe. Per tenere il conto degli oggetti `Person` creati utilizziamo la variabile statica di nome `personCount`. Dichiariamo infine gli attributi `ID` e `destinationFloor` come dati membro costanti.

```
829   // person.h
830   // Definizione della classe Person
831   #ifndef PERSON_H
832   #define PERSON_H
833
834   class Floor; // dichiarazione anticipata
835   class Elevator; // dichiarazione anticipata
836
837   class Person {
838
839   public:
840        Person( const int ); // costruttore
841        ~Person(); // distruttore
842        int getID() const; // restituisce l'ID della persona
843
844        void stepOntoFloor( Floor & );
845        void enterElevator( Elevator &, Floor & );
846        void exitElevator( const Floor &, Elevator & ) const;
847
848   private:
849        static int personCount; // numero totale delle persone
850        const int ID; // ID univoco di una persona
851        const int destinationFloor; // n° del piano di destinazione
852   );
853
854   #endif // PERSON_H
```

Figura 7.32 File di intestazione della classe Person.

L'implementazione della classe `Person` (Figura 7.33) inizia con il costruttore (linee 866-868), che riceve come solo argomento il piano di destinazione. Esso sarà utilizzato nell'output della simulazione. Il distruttore (linee 870-875) visualizza un messaggio che indica che la persona è uscita dall'ascensore.

```
855   // person.cpp
856   // Definizioni delle funzioni membro della classe Person.
857   #include <iostream.h>
858
859   #include "person.h"
860   #include "floor.h"
861   #include "elevator.h"
862
863   // inizializza il dato static member personCount
864   int Person::personCount = 0;
865
866   Person::Person( const int destFloor ) // costruttore
867        : ID( ++personCount ), destinationFloor( destFloor )
868   {}
869
870   Person::~Person() // distruttore
871   {
872        cout << "person " << ID << " exits simulation on floor "
873             << destinationFloor << " (person destructor invoked)"
874             << endl;
875   }
876
877   int Person::getID() const { return ID; } // leggi l'ID
878
879   // la persona accede al piano
880   void Person::stepOntoFloor( Floor& floor )
881   {
882        // notifica al piano l'arrivo di una persona
883        cout << "person " << ID << " steps onto floor "
884             << floor.getNumber() << endl;
885        floor.personArrives( this );
886
887        // preme il pulsante esterno del piano
888        cout << "person " << ID
889             << " presses floor button on floor "
890             << floor.getNumber() << endl;
891        floor.floorButton.pressButton();
892   }
893
894   // la persona entra nell'ascensore
895   void Person::enterElevator( Elevator &elevator, Floor &floor )
896   {
897        floor.personBoardingElevator();//la per. abbandona il piano
```

Figura 7.33 File di implementazione della classe Person (continua)

```
898
899       elevator.passengerEnters( this ); // la persona entra
900                                         // nell'ascensore
901       // preme il pulsante interno
902       cout << "person " << ID
903            << " presses elevator button" << endl;
904            elevator.elevatorButton.pressButton();
905    }
906
907    // la persona esce dall'ascensore
908    void Person::exitElevator(
909       const Floor &floor, Elevator &elevator ) const
910    {
911       cout << "person " << ID << " exits elevator on floor "
912            << floor.getNumber() << endl;
913       elevator.passengerExits();
914    }
```

Figura 7.33 File di implementazione della classe Person .

La funzione membro `stepOntoFloor` (linee 879–892) notifica al piano che è arrivata una persona che vuole iniziare una nuova corsa, inviandogli il messaggio `personArrives` (linea 885). La persona chiama quindi il metodo `pressButton` di `floorButton` (linea 891), che chiama l'ascensore..

La funzione membro `enterElevator` notifica al piano che la persona sta entrando nell'ascensore, inviandogli un messaggio `personBoardingElevator` (linea 897). La persona indica all'ascensore che vi sta entrando, inviandogli il messaggio `passengerEnters` (linea 899), e fa partire la corsa inviando il messaggio `pressButton` all'oggetto `elevatorButton`, (linea 904). La funzione membro `exitElevator` visualizza un messaggio che indica che la persona sta uscendo dall'ascensore, ed invia quindi il messaggio `passengerExits` ad `elevator`.

Possiamo considerare conclusa, a questo punto; la fase di implementazione del simulatore di ascensore. La prossima sezione "Pensare in termini di oggetti" non si troverà nel prossimo capitolo ma nel Capitolo 9, in cui discuteremo l'ereditarietà e la applicheremo al programma che abbiamo sviluppato.

Esercizi di autovalutazione

7.1 Completate le seguenti affermazioni:

a) La sintassi dell'_____ serve a inizializzare i membri costanti di una classe.

b) Una funzione non membro deve essere dichiarata come _____ di una classe per poter accedere ai dati `private` di tale classe.

c) L'operatore _____ alloca dinamicamente la memoria per un oggetto di un determinato tipo e restituisce un _____ a quel tipo.

d) Un oggetto costante deve essere _____ e non può essere modificato dopo essere stato creato.

e) Un dato membro _____ rappresenta un'informazione da condividere su tutta la classe.

f) Le funzioni membro non `static` di un oggetto possono accedere a un "autopuntatore" all'oggetto chiamato puntatore _____.

g) La parola chiave _____ specifica che un oggetto o una variabile non è modificabile dopo la sua inizializzazione.

h) Se non si fornisce un inizializzatore di membro per un oggetto membro di una classe, viene chiamato il _____ della classe.

i) Una funzione membro puo essere dichiarata come `static` se non accede ai dati membro _____ della classe.

j) Gli oggetti membro sono costruiti _____ dell'oggetto della classe che li contiene.

k) L'operatore _____ restituisce al sistema la memoria precedentemente allocata da new.

7.2 Individuate l'errore/gli errori e spiegate come correggerli.

a)
```
class Example {
public:
   Example( int y = 10 ) { data = y; }
   int getIncrementedData() const { return ++data; }
   static int getCount()
   {
    cout << "Data is " << data << endl;
     return count;
   }
private:
   int data;
   static int count;
};
```

b)
```
char *string;
string = new char[ 20 ];
free( string );
```

Risposte agli esercizi di autovalutazione

7.1 a) Inizializzatore di membro. b) `friend`. c) `new`, puntatore. d) inizializzato. e) `static`. f) `this`. g) `const`. h) costruttore di default. i) non `static`. j) prima. k) `delete`.

7.2 a) Errore: La definizione della classe `Example` contiene due errori. Il primo si trova nella funzione `getIncrementedData`. La funzione è dichiarata come `const`, ma modifica l'oggetto.
Correzione: Per il primo errore, eliminare la parola chiave `const` dalla definizione di `getIncrementedData`.
Errore: Il secondo errore si trova nella funzione `getCount`. La funzione è dichiarata come `static`, per cui non può accedere a membri della classe che non sono `static`.
Correzione: Per il secondo errore, eliminate la linea di output dalla definizione di `getCount`.

b) Errore: La memoria allocata da `new` è restituita al sistema chiamando la funzione della libreria standard del C `free`.
Correzione: Utilizzate in coppia gli operatori del C++ `new` e `delete` o le funzioni in stile C `malloc` e `free`. Non confondete mai le coppie.

Esercizi

7.3 Confrontate la coppia di operatori per l'allocazione dinamica `new` e `delete` con le funzioni della libreria standard del C `malloc` e `free`.

7.4 Spiegate il concetto di classe e funzione `friend` in C++.

7.5 La definizione della classe `Time` può contenere entrambi i costruttori che seguono? Se no, spiegate perché.

```
Time ( int h = 0, int m = 0, int s = 0 );
Time();
```

7.6 Che cosa accade se si specifica il tipo di dato restituito da un costruttore o da un distruttore (anche se `void`)?

7.7 Create la classe `Date` secondo queste specifiche:

a) Effettua l'output della data secondo più formati, come

```
GGG AAAA
GG/MM/AA
14 Giugno 1992
```

b) Usa costruttori in overloading per creare oggetti `Date` inizializzati con dati secondo i formati del punto (a).

c) Ha un costruttore che legge la data dal calendario di sistema tramite le funzioni della libreria standard del file `time.h` e imposta i suoi membri.

Nel Capitolo 8 saremo in grado di creare degli operatori per confrontare due date, e anche per determinare se una data è precedente a un'altra.

7.8 Create la classe `SavingsAccount` (conto corrente). Il tasso di interesse annuale è contenuto nel dato membro statico `annualInterestRate` per ciascun risparmiatore. Ogni membro della classe contiene il dato membro `private savingsBalance` che indica la somma correntemente in deposito. Scrivete la funzione membro `calculateMonthlyInterest` che calcola l'interesse mensile moltiplicando il bilancio per `annualInterestRate` diviso 12; questo interesse deve poi essere sommato a `savingsBalance` (bilancio del conto). Scrivere una funzione membro `static modifyInterestRate` che imposta il membro `static annualInterestRate` su un nuovo valore. Scrivete un programma di prova che faccia uso della classe `SavingsAccount`. Istanziate due oggetti `savingsAccount` distinti, `saver1` e `saver2`, con i bilanci di $2000.00 e $3000.00 rispettivamente. Impostate `annualInterestRate` al 3%, quindi calcolate l'interesse mensile e visualizzate il nuovo bilancio dei due risparmiatori. Infine impostate `annualInterestRate` al 4% e calcolate l'interesse del mese successivo, scrivendo ancora il nuovo bilancio dei due risparmiatori.

7.9 Create la classe `IntegerSet` (insieme di interi). Ogni oggetto di `IntegerSet` può memorizzare interi tra 0 e 100. Un insieme è rappresentato internamente da un array di uno e zero. L'elemento dell'array `a[ i ]` è 1 se l'intero `i` è contenuto nell'insieme. L'elemento dell'array `a[ j ]` è 0 se l'intero `j` non è contenuto nell'insieme. Il costruttore di default inizializza un insieme all'insieme vuoto, il cui array contiene soltanto zero.
Scrivete le funzioni membro che effettuano le comuni operazioni sugli insiemi. Per esempio, scrivete la funzione `unionOfIntegerSets` che crea un terzo insieme che è l'unione di due insiemi esistenti: un elemento del terzo insieme è 1 se è presente in almeno uno dei due insiemi origine, è 0 altrimenti.
Scrivete la funzione membro `intersectionOfIntegerSets` che crea un terzo insieme che è l'intersezione di due insiemi esistenti: un elemento del terzo insieme è 1 se è presente in entrambi gli insiemi origine, è 0 altrimenti.

Scrivete la funzione membro `insertElement` che inserisce un nuovo intero `k` nell'insieme (impostando `a[k]` a 1). Scrivete la funzione `deleteElement` che elimina dall'insieme l'intero `m` (impostando `a[m]` a 0).
Scrivete la funzione membro `setPrint` che visualizza un insieme come lista di numeri separati da spazi. Visualizzate soltanto gli elementi contenuti nell'insieme (la cui posizione, cioè, vale 1 nell'array). Visualizzate — per l'insieme vuoto.
Scrivete la funzione membro `isEqualTo` che determina se due insiemi sono uguali.
Scrivete un altro costruttore che riceve cinque argomenti interi e inizializza un oggetto insieme. Se desiderate fornire meno di cinque elementi al costruttore, prevedete argomenti di default tutti uguali a -1.
Ora scrivete un programma di prova che faccia uso della classe `IntegerSet`. Istanziate diversi oggetti `IntegerSet`. Verificate la correttezza di tutte le funzioni membro.

7.10 Sarebbe perfettamente comprensibile se la classe `Time` in Figura 7.8 rappresentasse l'orario internamente come il numero di secondi trascorso dall'ultima mezzanotte, anziché con i tre valori interi `hour`, `minute` e `second`. I client dovrebbero utilizzare gli stessi metodi pubblici e ottenere gli stessi risultati. Modificate la classe `Time` in Figura 7.8 per implementare `Time` come numero di secondi trascorsi dall'ultima mezzanotte, e dimostrare che i client non esperiranno alcuna differenza nella funzionalità della classe.

CAPITOLO 8

L'overloading degli operatori

Obiettivi

- Imparare a ridefinire gli operatori tramite l'overloading, in modo che possano operare su nuovi tipi di dato.
- Imparare a converture un oggetto di una classe in oggetto di un'altra classe.
- Conoscere le situazioni in cui è appropriato effettuare l'overloading degli operatori e le situazioni in cui non lo è.
- Studiare alcuni esempi interessanti di classi che utilizzano l'overloading degli operatori.
- Progettare le classi `Array`, `String` e `Date`.

8.1 Introduzione

Nei Capitoli 6 e 7 abbiamo introdotto i concetti fondamentali delle classi e il concetto di tipo di dato astratto (ADT). Le varie manipolazioni sugli oggetti di una classe (cioè sulle istanze dell'ADT) sono effettuate con l'invio di messaggi agli oggetti, nella forma di chiamate di funzione. Per alcuni tipi di classe la notazione di chiamata di funzione può rivelarsi scomoda e piuttosto pesante, pensiamo specialmente alle classi matematiche. Per queste classi sarebbe molto più vantaggioso poter sfruttare il ricco insieme di operatori predefiniti del C++ ed in questo capitolo vi mostriamo come ridefinire gli operatori in modo che possano operare sugli oggetti delle nostre classi.

Per poter raggiungere questo risultato occorre effettuare il cosiddetto *overloading degli operatori*. In C++ questa operazione è naturale e diretta, ma bisogna effettuarla con grande attenzione, perché overloading inappropriati e indiscriminati possono peggiorare la leggibilità del programma finale.

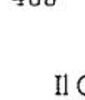
Un esempio di overloading di un operatore in C++ è l'operatore **<<**: esso ha diverse funzioni come l'inserimento di un dato nello stream di output o lo shifting a sinistra dei bit di un valore numerico. Lo stesso dicasi per l'operatore **>>** , infatti le sue funzioni sono normalmente l'estrazione di un dato dallo stream di input e lo shifting a destra dei bit. Nella libreria delle classi del C++ questi due operatori sono in overloading. Lo stesso linguaggio effettua l'overloading sugli operatori **+** e **-**: infatti essi possono effettuare diverse operazioni che dipendono dal contesto in cui compaiono, per esempio l'aritmetica intera, o a virgola mobile o sui puntatori.

Il C++ consente di effettuare l'overloading della maggior parte degli operatori, in modo da renderli sensibili al contesto in cui compaiono: il compilatore genera il codice appropriato sulla base del contesto e del modo in cui viene utilizzato l'operatore. Alcuni operatori sono candidati ad essere ridefiniti più frequentemente di altri come ad esempio l'operatore di assegnamento e gli operatori aritmetici + e - Il compito effettuato dall'overloading di un operatore può essere effettuato in modo equivalente da una chiamata di funzione esplicita, ma la notazione che fa uso degli operatori è generalmente più leggibile e significativa. In questo capitolo vedremo quali sono i casi in cui è conveniente utilizzare l'overloading degli operatori, spiegheremo come effettuarlo e vi presenteremo una serie di programmi completi che ne fanno uso.

8.2 L'overloading degli operatori: concetti fondamentali

La programmazione in C++ si concentra sulla creazione e sulla manipolazione dei tipi di dato costruiti a partire dai tipi predefiniti. Su questi ultimi possono essere utilizzati gli operatori del C++ che consentono di utilizzare una notazione concisa per esprimere le manipolazioni da effettuare.

La potenzialità del C++ consiste nel fatto che si possono utilizzare gli operatori anche sui nuovi tipi definiti dall'utente; anche se non è possibile creare nuovi operatori, è possibile effettuare l'overloading di quelli già esistenti, in modo tale che possano essere utilizzati sugli oggetti delle nuove classi e si comportino di volta in volta nel modo più appropriato.

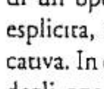


Ingegneria del software 8.1

L'overloading degli operatori è una delle caratteristica più importanti del linguaggio; essa contribuisce a rendere il C++ un linguaggio estensibile.

Buona abitudine 8.1

Se sostituite le funzioni con la ridefinizione di operatori che effettuano le stesse operazioni, migliorerete la leggibilità dei programmi.



Buona abitudine 8.2

Evitate gli eccessi: l'abuso dell'overloading di operatori può rendere un programma incomprensibile.

Il concetto di overloading di un operatore può sembrare una funzionalità quasi esotica, ma tenete presente che tutti i programmatori lo utilizzano regolarmente in modo implicito usando, per esempio, l'operatore di addizione (+) che opera in modo diverso su valori `int`, `float` e `double`. Ciò non toglie che esso possa operare correttamente su tutti e tre i tipi, e su molti altri tipi predefiniti, perché questo operatore è definito in overloading nello stesso linguaggio C++.

Per effettuare l'overloading di un operatore è sufficiente utilizzare una normale definizione di funzione nella quale il nome delle funzione è composto dalla parola chiave `operator` seguita dal simbolo dell'operatore. Ad esempio, nel caso dell' addizione (+) il nome della funzione è `operator+`.

Perché un operatore possa operare sugli oggetti di una classe, deve necessariamente essere ridefinito, con due importanti eccezioni. L'operatore di assegnamento (=) si può utilizzare su qualsiasi classe senza overloading esplicito: il suo comportamento di default è l'assegnamento membro a membro dei dati della classe. Tuttavia vedremo come questo comportamento così intuitivo può essere pericoloso se la classe contiene dei membri puntatore: in questi casi saremo costretti a effettuare un overloading esplicito dell'operatore di assegnamento. Anche l'operatore indirizzo (&) si può utilizzare su oggetti di qualsiasi classe senza overloading: il suo unico compito, infatti, è restituire l'indirizzo dell'area di memoria occupata dall'oggetto. Ad ogni modo, laddove occorra, anche questo operatore può essere ridefinito.

L'overloading è di una certa utilità soprattutto per le classi matematiche: esse hanno bisogno in genere di un ricco insieme di operatori che trattino i dati in modo coerente e significativo, rispecchiando la teoria matematica che li definisce. Sarebbe curioso, per esempio, nel caso dei numeri complessi effettuare la ridefinizione del solo operatore di addizione: su di essi, infatti, si utilizzano generalmente anche tutti gli altri operatori aritmetici.

Fortunatamente, il C++ abbonda di operatori; il riuscire a comprendere il significato di un operatore e il contesto appropriato in cui si utilizza è fondamentale per la decisione di effettuarne la ridefinizione allo scopo di farlo operare sulle vostre classi.

Lo scopo principale dell'overloading degli operatori è poter scrivere le espressioni in cui compaiono oggetti con la stessa concisione tipica delle espressioni in cui compaiono i tipi predefiniti. L'overloading, però, non è una funzionalità automatica: il programmatore ha la responsabilità di ridefinire gli operatori in modo che effettuino le operazioni desiderate. A volte la scelta migliore sta nel rendere gli operatori ridefiniti delle funzioni membro, altre volte conviene renderli funzioni `friend` e altre ancora (poche) non conviene renderli né membro né `friend`.

Fate attenzione a non utilizzare la ridefinizione in modo inappropriato e in contesti in cui ciò possa risultare controintuitivo: per esempio, non effettuate l'overloading dell'operatore + per effettuare operazioni paragonabili alla sottrazione, o quello dell'operatore / per effettuare operazioni paragonabili alla moltiplicazione. In questi casi i programmi risultanti rischiano di essere davvero incomprensibili.

Buona abitudine 8.3

Gli operatori ridefiniti dovrebbero operare sugli oggetti delle vostre classi in modo simile, se non identico, a come gli operatori originali agiscono sui tipi predefiniti. Evitatene un uso controintuitivo.

Buona abitudine 8.4

Prima di effettuare l'overloading di un operatore, consultate i manuali del vostro compilatore e accertatevi che non presenti particolari restrizioni o requisiti per l'operatore che avete scelto.

8.3 Restrizioni

È possibile effettuare l'overloading della maggior parte degli operatori del C++ che elenchiamo in Figura 8.1. In Figura 8.2, invece, elenchiamo gli operatori che non è possibile ridefinire.

Errore tipico 8.1

Se tentate di effettuare l'overloading di un operatore che non può essere ridefinito, commettete un errore di sintassi.

L'overloading non influisce sulla precedenza di un operatore e dunque bisogna porre attenzione perché l'operatore può essere utilizzato in contesti in cui la sua precedenza può sembrare inadatta, o complicare notevolmente la notazione. Ricordate comunque che in ogni caso potete ricorrere alle parentesi per forzare l'ordine di valutazione di un'espressione che contiene operatori in overloading.

Operatori che possono essere ridefiniti

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`
`~`	`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`
`/=`	`%=`	`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`
`--`	`->*`	`,`	`->`	`[]`	`()`	`new`	`delete`
`new[]`	`delete[]`						

Figura 8.1 Operatori che possono essere ridefiniti.

Operatori che non possono essere ridefiniti

`.`	`.*`	`::`	`?:`	`sizeof`

Figura 8.2 Operatori che non possono essere ridefiniti.

Ricordate che la ridefinizione di un operatore non modifica la sua associatività.

Un'altra restrizione riguarda il numero degli operandi: neanche questo è modificabile. Gli operatori unari restano unari anche nelle versioni ridefinite e lo stesso accade per quelli binari. L'unico operatore ternario del C++ (`?:`) non può essere ridefinito (Figura 8.2). Gli operatori `&`, `*`, `+` e `-` hanno ciascuno una versione unaria e una binaria e le due versioni possono essere ridefinite separatamente.

Fra le altre restrizioni ricordiamo che non è possibile creare nuovi operatori: si può effettuare soltanto l'overloading di operatori già esistenti. Purtroppo ciò impedisce di introdurre e utilizzare notazioni molto comuni, come la notazione `**` del BASIC per la funzione esponenziale.

Errore tipico 8.2

Se tentate di creare un nuovo operatore commettete un errore di sintassi.

Il significato dell'operazione compiuta da un operatore sui tipi predefiniti non si può modificare con l'overloading. Non è possibile, per esempio, cambiare il modo in cui + somma due `int`. L'overloading degli operatori può essere effettuato solo per le operazioni sui nuovi tipi definiti dall'utente o per le operazioni che coinvolgono nuovi tipi definiti dall'utente e tipi predefiniti.

Errore tipico 8.3

Se tentate di modificare il modo in cui un operatore opera sui tipi di dato predefiniti commettete un errore di sintassi.

Ingegneria del software 8.2

Almeno uno degli argomenti della funzione operatore deve essere un oggetto di una classe o un riferimento a un oggetto di una classe. Ricordando questa regola eviterete di cambiare il modo in cui un operatore opera sugli oggetti dei tipi predefiniti.

Se effettuate l'overloading dell'operatore di assegnamento, in modo da poter scrivere un'istruzione come questa

```
object2 = object2 + object1;
```

non otterrete implicitamente l'overloading, ad esempio, dell'operatore +=. Per cui non potrete scrivere l'istruzione

```
object2 += object1;
```

Per farlo, dovreste effettuare esplicitamente l'overloading dell'operatore +=.

Errore tipico 8.4

Se effettuate l'overloading di un operatore come + non assumete che il compilatore provveda automaticamente all'overloading di tutti gli operatori correlati, come +=. Gli operatori vanno ridefiniti esplicitamente.

Errore tipico 8.5

Gli operatori unari restano unari, e quelli binari restano binari anche dopo l'overloading: se tentate di cambiare il numero di operandi di un operatore commettete un errore di sintassi.

Buona abitudine 8.5

Per ottenere un comportamento coerente di operatori correlati, conviene effettuare l'overloading di uno di essi e implementare gli altri in funzione del primo: per esempio, utilizzate l'overloading di + per implementare l'overloading di +=.

8.4 La progettazione delle funzioni operatore: funzioni membro o funzioni friend?

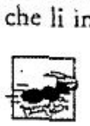
Le funzioni operatore possono essere funzioni membro o meno; nel caso si tratti di funzioni non membro esse, generalmente, sono rese `friend` per motivi di efficienza. Le funzioni membro utilizzano implicitamente il puntatore `this` per ottenere uno degli argomenti, corrispondente a un oggetto di una classe.

Tale argomento (l'oggetto della classe) deve invece comparire esplicitamente nella lista degli argomenti quando si definisce una funzione non membro. Se si effettua l'overloading di `()`, `[]`, `->` o di un qualsiasi operatore di assegnamento, la funzione operatore deve essere dichiarata come membro della classe. Per quanto riguarda tutti gli altri operatori, invece, le funzioni operatore possono anche essere funzioni non membro.

Indipendentemente da come decidete di implementare la funzione di overloading, membro o non membro, l'operatore verrà utilizzato nelle espressioni allo stesso modo. A questo punto è lecito chiedersi quale sia l'implementazione migliore

Se una funzione operatore è implementata come funzione membro, l'operando sinistro (o l'unico previsto) deve essere un oggetto o un riferimento a un oggetto della classe dell'operatore. Se si ha bisogno di porre nell'operando sinistro un oggetto diverso, la funzione operatore deve essere necessariamente implementata come funzione non membro. Daremo un esempio di ciò nella Sezione 8.5, in cui effettuiamo l'overloading di `<<` e `>>` rispettivamente come operatore di inserimento nello stream e di estrazione dallo stream. Per accedere direttamente ai membri `private` e `protected` della classe, la funzione operatore deve essere dichiarata come `friend`.

L'overloading dell'operatore `<<` ha un operando sinistro di tipo `ostream &` (ad esempio `cout` nell'espressione `cout << classObject`), per cui deve essere definita come funzione non membro. Allo stesso modo, l'overloading dell'operatore `>>` ha un operando sinistro di tipo `istream &` (`cin` nell'espressione `cin >> classObject`), per cui anch'esso deve essere una funzione non membro. Questi due operatori, inoltre, devono poter accedere ai dati membro `private` dell'oggetto in output o in input, per cui a volte le funzioni operatore che li implementano sono dichiarate come `friend` per motivi di efficienza.



Obiettivo efficienza 8.1

Se una funzione operatore non è funzione membro né funzione `friend` *di una classe, può accedere ai dati* `private` *o* `protected` *della classe soltanto tramite le funzioni* set *o* get *eventualmente presenti nell'interfaccia pubblica della classe. La chiamata a queste funzioni ha un costo in termini di efficienza, per cui esse dovrebbero essere rese funzioni* `inline`*, per migliorarne le prestazioni.*

Un operatore ridefinito come membro di una data classe viene chiamato soltanto quando l'operando sinistro, nel caso di operatore binario, o l'unico operando, nel caso di operatore unario, è un oggetto di tale classe.

Un'altra ragione per scrivere una funzione operatore come funzione non membro è consentire all'operatore di essere commutativo. Per esempio, supponiamo di avere un oggetto `number` di tipo `long int` e un oggetto `bigInteger1` della classe `HugeInteger`, che è una classe di interi di dimensione arbitraria, senza alcuna restrizione imposta dall'hardware della macchina (implementeremo la classe `HugeInteger` negli esercizi). L'operatore di addizione (+) produce un oggetto `HugeInteger` temporaneo per la somma di un `HugeInteger` e un `long int` (come in `bigInteger1 + number`), o per la somma di un `long int` e un `HugeInteger` (come in `number + bigInteger1`). Abbiamo bisogno che l'operazione di addizione sia commutativa, proprio come accade nella realtà Il problema è che se la funzione operatore è una funzione membro, l'oggetto della classe deve comparire a sinistra dell'operatore di addizione. Perciò scegliamo di effettuare l'overloading dell'operatore come funzione non membro `friend`, in modo che `HugeInteger` possa comparire anche come

operando destro dell'addizione. Ad ogni modo la funzione `operator+` che riceve `HugeInteger` come operando sinistro può essere benissimo funzione membro. Tenete a mente che una funzione non membro non deve necessariamente essere `friend` se l'interfaccia pubblica della classe prevede funzioni *set* e *get*, specialmente se queste sono `inline`.

8.5 L'overloading degli operatori di inserimento/estrazione per l'I/O su stream

Il C++ effettua l'input e l'output dei tipi predefiniti tramite l'operatore di inserimento nello stream `>>` e l'operatore di estrazione dallo stream `<<`. Questi due operatori sono ridefiniti (nelle librerie di classi fornite con i compilatori C++) in modo da trattare tutti i tipi predefiniti in stile C come `char *`, le stringhe e i puntatori. Gli operatori di inserimento ed estrazione possono subire un'ulteriore overloading per l'input e l'output dei tipi definiti dall'utente; il programma in Figura 8.3 mostra l'overloading di questi due operatori per la classe `PhoneNumber`, che rappresenta numeri telefonici. Il programma assume che i numeri siano immessi correttamente. Negli esercizi vi chiederemo di aggiungere qualche controllo sulla validità dei dati.

```
1  // Fig. 8.3: fig08_03.cpp
2  // Overloading degli operatori di inserimento nello stream
3  // e di estrazione dallo stream.
4  #include <iostream.h>
5  #include <iomanip.h>
6
7  class PhoneNumber {
8     friend ostream &operator<<( ostream&, const PhoneNumber & );
9     friend istream &operator>>( istream&, PhoneNumber & );
10
11 private:
12    char areaCode[ 4 ];  // prefisso a 3 cifre e terminatore nullo
13    char exchange[ 4 ];  // prima parte (3 cifre)del numero e
14                         // terminatore nullo
15    char line[ 5 ];      // seconda parte del numero (4 cifre) e
16                         // terminatore nullo
17 };
18
19 // Overloading dell'operatore di inserimento (non può essere
20 // funzione membro se vogliamo invocarlo nella forma
21 // cout << somePhoneNumber;).
22 ostream &operator<<( ostream &output, const PhoneNumber &num )
23 {
24    output << "(" << num.areaCode << ") "
25           << num.exchange << "-" << num.line;
26    return output;     // consente di scrivere cout << a << b << c;
27 }
28
```

Figura 8.3 Inserimento nello stream ed estrazione dallo stream per un tipo definito dall'utente (cotinua)

```
29  istream &operator>>( istream &input, PhoneNumber &num )
30  {
31     input.ignore();                    // ignora la (
32     input >> setw( 4 ) >> num.areaCode; // input del prefisso
33     input.ignore( 2 );                 // ignora la ) e lo spazio
34     input >> setw( 4 ) >> num.exchange; // input della prima parte
35                                        // del numero
36     input.ignore();                    // ignora il -
37     input >> setw( 5 ) >> num.line;    // input della seconda
38                                        // parte del numero
39     return input;      // consente di scrivere cin >> a >> b >> c;
40  }
41
42  int main()
43  {
44     PhoneNumber phone; // crea l'oggetto phone
45
46     cout << "Enter phone number in the form (123) 456-7890:\n";
47
48     // cin >> phone invoca operator>>
49     // nella chiamata operator>>( cin, phone ).
50     cin >> phone;
51
52     // cout << phone invoca operator<<
53     // nella chiamata operator<<( cout, phone ).
54     cout << "The phone number entered was: " << phone << endl;
55     return 0;
56  }
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Figura 8.3 Inserimento nello stream ed estrazione dallo stream per un tipo definito dall'utente.

La funzione operatore di estrazione `operator>>` (linea 29) riceve come argomenti un riferimento a `istream` di nome `input` e un riferimento a un oggetto `PhoneNumber` di nome `num`, e restituisce un riferimento a `istream`. La funzione `operator>>` esegue l'input dei numeri telefonici nel formato

```
(800) 555-1212
```

e scrive tali numeri negli oggetti della classe `PhoneNumber`. Quando il compilatore incontra l'espressione

```
cin >> phone
```

in `main`, genera la chiamata di funzione

```
operator>>( cin, phone );
```

Quando viene eseguita questa chiamata, il parametro di riferimento `input` diventa un alias di `cin` e il parametro di riferimento `num` diventa un alias di `phone`. La funzione

operatore legge come stringhe le tre parti del numero di telefono e le memorizza nei membri `areaCode`, `exchange` e `line` del riferimento all'oggetto `PhoneNumber` (`num` nella funzione operatore e `phone` in `main`). Il manipolatore dello stream `setw` assicura che in ogni array di caratteri venga memorizzato il numero corretto di caratteri.

Ricordate che, usato con `cin`, `setw` limita il numero di caratteri da leggere al numero specificato dal suo argomento meno uno, per esempio `setw( 4 )` consente la lettura e il salvataggio di tre caratteri, riservando lo spazio anche per il terminatore nullo. Parentesi, spaziature e trattini sono ignorati dalla funzione `ignore` membro di `istream`, che scarta il numero specificato di caratteri dallo stream di input (un carattere per default). La funzione `operator>>` restituisce `input` (cioè `cin`), che è un riferimento a `istream`. Ciò consente di effettuare operazioni di input a cascata, prima di oggetti `PhoneNumber` e poi di altri oggetti, siano anch'essi di tipo `PhoneNumber` o di altro tipo. Per esempio, l'input di due oggetti `PhoneNumber` si può effettuare così:

```
cin >> phone1 >> phone2;
```

Viene valutata per prima l'espressione `cin >> phone1` che effettua la chiamata a

```
operator>>( cin, phone1 );
```

Questa chiamata restituisce un riferimento a `cin` come valore di `cin >> phone1`, per cui il resto dell'espressione originaria viene poi interpretato semplicemente come `cin >> phone2`. Successivamente, dunque, viene effettuata la chiamata a

```
operator>>( cin, phone2 );
```

L'operatore di inserimento riceve come argomenti un riferimento a `ostream` (`output`) e un riferimento (`num`) a un tipo definito dall'utente (`PhoneNumber`), e restituisce un riferimento a `ostream`. La funzione `operator<<` visualizza gli oggetti `PhoneNumber`. Quando il compilatore incontra l'espressione

```
cout << phone
```

in `main`, genera la chiamata alla funzione non membro

```
operator<<( cout, phone );
```

La funzione `operator<<` visualizza le parti di un numero telefonico come stringhe, perché è questo il formato in cui sono memorizzate.

Osservate che le funzioni `operator>>` e `operator<<` sono dichiarate nella classe `PhoneNumber` come funzioni non membro e `friend`. La necessità di renderle non membro nasce dal fatto che l'oggetto della classe `PhoneNumber` compare in ogni caso come operando destro dell'operatore: l'operando della classe dovrebbe comparire a sinistra dell'operatore in una funzione membro. Gli overloading degli operatori di input/output sono dichiarati come `friend` se devono accedere ai membri non `public` della classe.

Notate inoltre che il riferimento a `PhoneNumber` nella lista dei parametri di `operator<<` è `const`, perché l'oggetto `PhoneNumber` è semplicemente inviato in output, mentre nella lista di parametri di `operator>>` non lo è, perché nell'oggetto `PhoneNumber` deve essere scritto il numero di telefono ricevuto in input.

Ingegneria del software 8.3

In C++ è possibile creare nuove funzionalità di input/output senza modificare le dichiarazioni e i membri `private` delle classi `ostream` e `istream`. Questo è un altro esempio dell'estensibilità del linguaggio.

8.6 L'overloading degli operatori unari

Un operatore unario può essere ridefinito come funzione membro non `static` senza argomenti o come funzione non membro che riceve un solo argomento: questo deve essere un oggetto o un riferimento a un oggetto della classe. Le funzioni membro che implementano gli operatori devono essere non `static` per poter accedere ai dati non `static` della classe. Ricordate che le funzioni membro `static` possono accedere soltanto ai dati membro `static` della classe. Nel seguito di questo capitolo effettueremo l'overloading dell'operatore unario `!` per verificare se un oggetto della classe `String` è vuoto (il risultato è di tipo `bool`). Nell'overloading di `!` come funzione membro non `static` senza argomenti, se `s` è un oggetto o un riferimento a un oggetto della classe `String`, l'espressione `!s` genera la chiamata a `s.operator!()`. L'operando `s` è l'oggetto della classe per cui viene invocata la funzione `operator!` della classe `String`. La dichiarazione della funzione nella definizione di classe è:

```
class String {
public:
   bool operator!() const;
   ...
};
```

Su un operatore unario come `!` si può effettuare l'overloading come funzione non membro con un argomento, in due modi diversi: con un argomento che è un oggetto (occorre una copia dell'oggetto, in modo che gli effetti collaterali della funzione non si ripercuotano sull'oggetto originario), o con un argomento che è un riferimento a un oggetto (senza copia dell'oggetto, per cui gli effetti collaterali della funzione si ripercuotono sull'oggetto originario). Se `s` è un oggetto o un riferimento a un oggetto della classe `String`, l'espressione `!s` viene trattata come chiamata a `operator!( s )`, che invoca la funzione non membro `friend` della classe `String` dichiarata qui di seguito:

```
class String {
   friend bool operator!( const String & );
   ...
};
```

Buona abitudine 8.6

Nell'effettuare l'overloading di un operatore unario, è preferibile rendere la funzione operatore membro della classe anziché non membro e `friend`*. Le funzioni e le classi* `friend` *sono da evitare, a meno che non siano strettamente necessarie. Funzioni e classi* `friend` *violano l'incapsulamento di una classe.*

8.7 L'overloading degli operatori binari

Un operatore binario può essere ridefinito come funzione membro non `static` con un argomento, o come una funzione non membro con due argomenti, uno dei quali deve essere un oggetto o un riferimento a un oggetto della classe. Nel seguito di questo capitolo effettueremo l'overloading di `+=` per concatenare due oggetti stringa. Ridefinendo l'operatore binario `+=` come funzione membro non `static` della classe `String` con un argomento, se `y` e `z` sono oggetti della classe `String`, `y += z` viene trattato come `y.operator+=( z )`, che invoca la funzione membro `operator+=` dichiarata qui di seguito:


```
class String {
public:
   const String &operator+=( const String & );
   ...
};
```

Se, invece, si sceglie di effettuare l'overloading dell'operatore `+=` come funzione non membro, essa deve ricevere due argomenti, uno dei quali deve essere un oggetto o un riferimento a un oggetto della classe. Se `y` e `z` sono oggetti o riferimenti a oggetti della classe `String`, l'espressione `y += z` è trattata come una chiamata a `operator+=( y, z )`, che invoca `operator+=`, funzione non membro e `friend` dichiarata qui di seguito:

```
class String {
   friend const String &operator+=( String &,
                                    const String & );
   ...
};
```

8.8 Progettazione della classe Array

La notazione degli array in C++ costituisce un'alternativa rispetto all'uso esplicito dei puntatori, per cui il potenziale di errori che comporta è piuttosto cospicuo. Per esempio, un programma può tranquillamente oltrepassare i limiti di un array, perché il C++ non effettua alcun controllo sulla validità degli indici. Gli indici di un array di n elementi sono forzati ad essere esattamente $0, \ldots, n-1$ e non è consentita un'indicizzazione alternativa. Un intero array non può essere ricevuto in input o inviato in output tutto insieme: occorre leggere o scrivere individualmente ogni elemento; due array non possono essere confrontati con gli operatori relazionali o di uguaglianza, per il fatto che il nome di un array è semplicemente un puntatore all'area di memoria occupata dal suo primo elemento. Quando un array viene passato a una funzione generica che opera su array di qualsiasi dimensione, dev'essere accompagnato da un secondo argomento che specifica le dimensioni dell'array passato. Un array non può essere assegnato a un altro array tramite gli operatori di assegnamento, perché il nome di un array è un puntatore costante (un *rvalue*), e dunque non può comparire sul lato sinistro di un assegnamento.

Nonostante queste limitazioni il C++ fornisce uno strumento utile per implementare tutte le caratteristiche che mancano agli array attraverso l'overloading degli operatori.

In questa sezione svilupperemo una classe array che effettua controlli sulla validità degli indici, in modo che non oltrepassino i limiti fisici dell'array. Sarà possibile inoltre assegnare un intero array a un altro con l'operatore di assegnamento. Gli oggetti di questa classe array conosceranno le proprie dimensioni, che quindi non dovranno essere passate alle funzioni in un argomento separato. Grazie agli operatori di inserimento/estrazione sarà possibile effettuare l'I/O di interi array e, inoltre, per confrontare due array si potranno utilizzare gli operatori `==` e `!=`. Infine, la nostra classe array conterrà un membro `static` che conta gli oggetti array istanziati in un dato momento nel programma.

Grazie a questo esempio riuscirete ad apprezzare appieno il concetto di astrazione dei dati. La progettazione di una classe è un'attività stimolante e creativa, se l'obiettivo è quello di creare classi che costituiscano un vero e proprio "patrimonio" software da cui attingere per scrivere i propri programmi.

Il programma in Figura 8.4 illustra la classe **Array** e l'overloading degli operatori associati. Prima analizziamo il programma di esempio in `main` ed in seguito prenderemo in considerazione la definizione della classe, le sue funzioni membro e le funzioni `friend`.

```
1   // Fig. 8.4: array1.h
2   // La semplice classe Array (di interi)
3   #ifndef ARRAY1_H
4   #define ARRAY1_H
5
6   #include <iostream.h>
7
8   class Array {
9      friend ostream &operator<<( ostream &, const Array & );
10     friend istream &operator>>( istream &, Array & );
11  public:
12     Array( int = 10 );                    // costruttore di default
13     Array( const Array & );               // costruttore di copia
14     ~Array();                             // distruttore
15     int getSize() const;                  // restituisce size
16     const Array &operator=( const Array & ); // assegna gli array
17     bool operator==( const Array & ) const;  // confronto
18
19     // Determina se due array non sono uguali e in tal caso
20     // restituisce true, altrimenti false (usa operator==).
21     bool operator!=( const Array &right ) const
22        { return ! ( *this == right ); }
23
24     int &operator[]( int );           // operatore di indicizzazione
25     const int &operator[]( int ) const; // oper. di indicizzazione
26     static int getArrayCount();       // Restituisce il numero
27                                       // di array istanziati.
28  private:
29     int size; // dimensione dell'array
30     int *ptr; // puntatore al primo elemento dell'array
31     static int arrayCount;  // numero di Array istanziati
32  };
33
34  #endif

35  // Fig 8.4: array1.cpp
36  // Definizioni delle funzioni membro della classe Array
37  #include <iostream.h>
38  #include <iomanip.h>
39  #include <stdlib.h>
40  #include <assert.h>
41  #include "array1.h"
42
43  //Inizializza il dato membro static nello scope a livello di file
44  int Array::arrayCount = 0;   // ancora nessun oggetto istanziato
```

Figura 8.4 La classe Array e e la ridefinizione degli operatori (continua)


```
45
46  // Costruttore di default della classe Array (size di default=10)
47  Array::Array( int arraySize )
48  {
49     size = ( arraySize > 0 ? arraySize : 10 );
50     ptr = new int[ size ]; // crea spazio per l'array
51     assert( ptr != 0 ); // termina se non è stata allocata memoria
52     ++arrayCount;          // c'è un nuovo oggetto istanziato
53
54     for ( int i = 0; i < size; i++ )
55        ptr[ i ] = 0;          // inizializza l'array
56  }
57
58  // Costruttore di copia per la classe Array deve ricevere
59  // un riferimento per non creare una ricorsione infinita
60  Array::Array( const Array &init ) : size( init.size )
61  {
62     ptr = new int[ size ];   // crea spazio per l'array
63     assert( ptr != 0 ); // termina se non è stata allocata memoria
64     ++arrayCount;            // c'è un nuovo oggetto istanziato
65
66     for ( int i = 0; i < size; i++ )
67        ptr[ i ] = init.ptr[ i ];  // copia init nell'oggetto
68  }
69
70  // Distruttore della classe Array
71  Array::~Array()
72  {
73     delete [] ptr;  // restituisce al sistema lo spazio dell'array
74     --arrayCount;             // c'è un oggetto in meno
75  }
76
77  // Ottiene size
78  int Array::getSize() const { return size; }
79
80  // Overloading dell'operatore di assegnamento
81  // il tipo restituito è const per evitare: ( a1 = a2 ) = a3
82  const Array &Array::operator=( const Array &right )
83  {
84     if ( &right != this ) {  // controlla che non si tratti di
85                              // autoassegnamento
86  // per array di dimensioni diverse, dealloca l'array originario
87  // a sinistra dell'operatore, e ne alloca uno nuovo.
88        if ( size != right.size ) {
89           delete [] ptr;      // restituisce al sistema lo spazio
90           size = right.size; // ridimensiona l'oggetto
91           ptr=new int[size]; //crea spazio per la copia dell'array
92           assert( ptr != 0 ); //termina se l'allocazione è fallita
93        }
```

Figura 8.4 La classe Array e e la ridefinizione degli operatori (continua)

```
94
95       for ( int i = 0; i < size; i++ )
96          ptr[ i ] = right.ptr[ i ]; // copia l'array nell'oggetto
97    }
98
99    return *this;   // consente di scrivere x = y = z;
100 }
101
102 // Determina se due array sono uguali e in tal caso
103 // restituisce true, altrimenti false.
104 bool Array::operator==( const Array &right ) const
105 {
106    if ( size != right.size )
107       return false;    // array di dimensioni diverse
108
109    for ( int i = 0; i < size; i++ )
110       if ( ptr[ i ] != right.ptr[ i ] )
111          return false; // gli array non sono uguali
112
113    return true;        // gli array sono uguali
114 }
115
116 // Overloading dell'operatore di indicizzazione per Array:
117 // la restituzione di un riferimento crea un lvalue
118 int &Array::operator[]( int subscript )
119 {
120    // controlla che l'indice sia valido
121    assert( 0 <= subscript && subscript < size );
122
123    return ptr[ subscript ]; // restituisce un riferimento
124 }
125
126 // Overloading dell'operatore di indicizzazione per Array:
127 // la restituzione di un riferimento costante crea un rvalue
128 const int &Array::operator[]( int subscript ) const
129 {
130    // controlla che l'indice sia valido
131    assert( 0 <= subscript && subscript < size );
132
133    return ptr[ subscript ]; // restituisce un riferimento const
134 }
135
136 // Restituisce il numero di oggetti Array istanziati:
137 // si noti che le funzioni static non possono essere const
138 int Array::getArrayCount() { return arrayCount; }
139
140 // Overloading dell'operatore di input per la classe Array
141 // effettua l'input dei valori per un intero array.
```

Figura 8.4 La classe Array e e la ridefinizione degli operatori (continua)


```
142 istream &operator>>( istream &input, Array &a )
143 {
144    for ( int i = 0; i < a.size; i++ )
145       input >> a.ptr[ i ];
146
147    return input;   // consente di scrivere cin >> x >> y;
148 }
149
150 // Overloading dell'operatore di output per la classe Array
151 ostream &operator<<( ostream &output, const Array &a )
152 {
153    int i;
154
155    for ( i = 0; i < a.size; i++ ) {
156       output << setw( 12 ) << a.ptr[ i ];
157
158       if ( ( i + 1 ) % 4 == 0 ) // 4 numeri per riga di output
159          output << endl;
160    }
161
162    if ( i % 4 != 0 )
163       output << endl;
164
165    return output;   // consente di scrivere cout << x << y;
166 }
167 // Fig. 8.4: fig08_04.cpp
168 // Programma di esempio per la semplice classe Array
169 #include <iostream.h>
170 #include "array1.h"
171
172 int main()
173 {
174    // non sono stati ancora istanziati oggetti
175    cout << "# of arrays instantiated = "
176         << Array::getArrayCount() << '\n';
177
178    // crea due array e visualizza count
179    Array integers1( 7 ), integers2;
180    cout << "# of arrays instantiated = "
181         << Array::getArrayCount() << "\n\n";
182
183    // visualizza dimensione e contenuto di integers1.
184    cout << "Size of array integers1 is "
185         << integers1.getSize()
186         << "\nArray after initialization:\n"
187         << integers1 << '\n';
188
```

Figura 8.4 La classe Array e e la ridefinizione degli operatori (continua)

```
189    // visualizza dimensione e contenuto di integers2
190    cout << "Size of array integers2 is "
191         << integers2.getSize()
192         << "\nArray after initialization:\n"
193         << integers2 << '\n';
194
195    // input e visualizzazione di integers1 e integers2
196    cout << "Input 17 integers:\n";
197    cin >> integers1 >> integers2;
198    cout << "After input, the arrays contain:\n"
199         << "integers1:\n" << integers1
200         << "integers2:\n" << integers2 << '\n';
201
202    // utilizza l'overloading dell'op. di disuguaglianza (!=)
203    cout << "Evaluating: integers1 != integers2\n";
204    if ( integers1 != integers2 )
205       cout << "They are not equal\n";
206
207    // crea array integers3 utilizzando integers1
208    // come inizializzatore; visualizza dimensione e contenuto
209    Array integers3( integers1 );
210
211    cout << "\nSize of array integers3 is "
212         << integers3.getSize()
213         << "\nArray after initialization:\n"
214         << integers3 << '\n';
215
216    // utilizza l'overloading  dell'operatore di assegnamento (=)
217    cout << "Assigning integers2 to integers1:\n";
218    integers1 = integers2;
219    cout << "integers1:\n" << integers1
220         << "integers2:\n" << integers2 << '\n';
221
222 // utilizza l'overloading dell'operatore di uguaglianza (==)
223    cout << "Evaluating: integers1 == integers2\n";
224    if ( integers1 == integers2 )
225       cout << "They are equal\n\n";
226 // utilizza l'overloading dell'operatore di
227 // indicizzazione per creare un rvalue
228    cout << "integers1[5] is " << integers1[5] << '\n';
229 // utilizza l'overloading dell'operatore di indicizzazione
230 // per creare un lvalue
231    cout << "Assigning 1000 to integers1[5]\n";
232    integers1[5] = 1000;
233    cout << "integers1:\n" << integers1 << '\n';
234
235    // tentativo di utilizzare un indice non valido
236    cout << "Attempt to assign 1000 to integers1[15]" << endl;
```

Figura 8.4 La classe Array e e la ridefinizione degli operatori (continua)


```
237    integers1[15] = 1000;  // ERRORE: fuori dei limiti consentiti
238
239    return 0;
240 }
```

```
# of arrays instantiated = 0
# of arrays instantiated = 2

Size of array integers1 is 7
Array after initialization:
        0       0       0       0
        0       0       0

Size of array integers2 is 10
Array after initialization:
        0       0       0       0
        0       0       0       0
        0       0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1:
        1       2       3       4
        5       6       7
integers2:
        8       9       10      11
        12      13      14      15
        16      17
Evaluating: integers1 != integers2
They are not equal

Size of array integers3 is 7
Array after initialization:
        1       2       3       4
        5       6       7

Assigning integers2 to integers1:
integers1:
        8       9       10      11
        12      13      14      15
        16      17
integers2:
        8       9       10      11
        12      13      14      15
        16      17
Evaluating: integers1 == integers2
They are equal
```

Figura 8.4 La classe Array e e la ridefinizione degli operatori (continua)

```
integers1[5] is 13
Assigning 1000 to integers1[5]
integers1:
         8         9        10        11
        12      1000        14        15
        16        17
Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
file Array1.cpp, line 82
abnormal program termination
```

Figura 8.4 La classe `Array` e e la ridefinizione degli operatori.

La variabile membro `static` `arrayCount` conta gli oggetti `Array` istanziati durante l'esecuzione del programma. Il programma inizia con la funzione membro `static` `getArrayCount` (linea 176) che restituisce il numero di array istanziati finora. Quindi il programma istanzia due oggetti `Array` (linea 179), `integers1` di sette elementi e `integers2` con la dimensione di default di 10 elementi (il valore di default è specificato nel costruttore di default di `Array`). La linea 181 chiama nuovamente `getArrayCount` per conoscere il valore di `arrayCount`.

Le linee 184-187 utilizzano la funzione membro `getSize` per determinare la dimensione di `integers1` e inviano in output `integers1` tramite l'*overloading dell'operatore di* inserimento, per confermare che il costruttore di `Array` ha inizializzato correttamente i due elementi. Successivamente, le linee 190-193 leggono e inviano in output nello stesso modo la dimensione di `integers2`.

All'utente viene chiesto di immettere 17 valori interi. Per leggere questi valori in entrambi gli array viene utilizzato l'overloading dell'operatore *di estrazione alla linea 197*

```
cin >> integers1 >> integers2;
```

I primi sette valori sono memorizzati in `integers1` e i restanti 10 in `integers2`. Nelle linee 198-200, i due array sono inviati in output, per confermare che l'input è stato eseguito correttamente.

La linea 204 verifica che l'overloading dell'operatore di disuguaglianza funzioni correttamente, valutando la condizione

```
integers1 != integers2
```

e il programma riporta che gli array effettivamente non sono uguali.

La linea 209 istanzia un terzo oggetto `Array` di nome `integers3` e lo inizializza con `integers1`. Questa operazione invoca il *costruttore di copia* di `Array`, che copia gli elementi di `integers1` in `integers3`. Parleremo tra breve dei dettagli di questo costruttore.

Le linee 211-214 inviano in output la dimensione di `integers3` e lo stesso `integers3` con l'operatore di inserimento, per confermare che il costruttore ha inizializzato correttamente gli elementi dell'array.

Quindi la linea 218 verifica che l'overloading dell'operatore di assegnamento (=) funzioni *correttamente con l'istruzione*

```
integers1 = integers2;
```

Entrambi gli `Array` sono visualizzati alle linee 219 e 220, per confermare che l'assegnamento è riuscito. È interessante notare che `integers1` conteneva originariamente 7 interi e dunque è stato ridimensionato per contenere i 10 elementi di `integers2`. Come vedremo, l'overloading dell'operatore di assegnamento effettua il ridimensionamento in maniera automatica e trasparente per la funzione chiamante.

Dopo di ciò, la linea 224 si serve dell'overloading dell'operatore di uguaglianza (==) per confermare che `integers1` e `integers2` sono identici dopo l'assegnamento.

La linea 228 utilizza l'overloading dell'operatore di indicizzazione per riferire `integers1[ 5 ]`, che è un elemento valido di `integers1`. Questo dato è utilizzato come *rvalue* per visualizzare il valore di `integers1[ 5 ]`. La linea 232 utilizza `integers1[ 5 ]` come *lvalue* per assegnare all'elemento 5 di `integers1` il valore 1000. Osservate che `operator[]` restituisce un riferimento che può essere utilizzato come *lvalue* dopo aver determinato che 5 è un elemento valido di `integers1`.

La linea 237 tenta di assegnare il valore 1000 a `integers1[ 15 ]`, che non è un elemento valido dell'array. L'overloading dell'operatore `[]` rileva questo errore e causa la terminazione del programma.

È interessante notare che l'operatore di indicizzazione `[]` può essere utilizzato per selezionare gli elementi di tutte le classi container ordinate, come le liste concatenate, le *stringhe, i dizionari e così via. Inoltre gli indici non devono necessariamente essere degli* interi: si possono utilizzare, per esempio, caratteri, stringhe, `float` e persino oggetti di classi definite dall'utente.

Ora che abbiamo visto come funziona il programma, osserviamo attentamente l'intestazione della classe e la definizione delle funzioni membro. Le linee 29-31

```
int size; // dimensione dell'array
int *ptr; // puntatore al primo elemento dell'array
static int arrayCount;  // numero di Array istanziati
```

rappresentano i dati membro `private` della classe. L'array è composto da un membro `size` che indica il numero di elementi dell'array, da un puntatore a un `int` (`ptr`) che punta all'area di memoria allocata dinamicamente destinata a contenere gli interi dell'oggetto `Array` e dal membro `static` `arrayCount` che indica il numero di oggetti `Array` istanziati dall'inizio del programma.

Le linee 9 e 10

```
friend ostream &operator<<( ostream &, const Array & );
friend istream &operator>>( istream &, Array & );
```

dichiarano la ridefinizione degli operatori di inserimento e di estrazione come `friend` della classe `Array`. Quando il compilatore incontra l'espressione

```
cout << arrayObject
```

invoca la funzione `operator<<( ostream &, const Array & )` generando la chiamata

```
operator<<( cout, arrayObject )
```

Quando il compilatore incontra l'espressione

```
cin >> arrayObject
```

invoca la funzione `operator>>( istream &, Array & )` generando la chiamata

```
operator>>( cin, arrayObject )
```

Vi facciamo notare ancora una volta che le funzioni operatore `<<` e `>>` non possono essere membri della classe `Array` perché l'oggetto `Array` viene menzionato sempre come operando destro per entrambi gli operatori. Se le rendessimo funzioni membro della classe `Array`, dovremmo necessariamente utilizzare una notazione un po' più scomoda e meno intuitiva per l'I/O di un `Array`:

```
arrayObject << cout;
arrayObject >> cin;
```

La funzione `operator<<` (definita alla linea 151) visualizza il numero di elementi indicato da `size` dell'array che si trova all'indirizzo `ptr`. La funzione `operator>>` (definita alla linea 142) riceve in input i valori e li memorizza direttamente nell'array a cui punta `ptr`. Entrambe le funzioni restituiscono un riferimento, in modo tale da poter scrivere istruzioni di I/O in cascata.

La linea

```
Array( int = 10 );                    // costruttore di default
```

dichiara il costruttore di default della classe e indica che il valore di default della dimensione dell'array è 10. Quando il compilatore incontra la dichiarazione

```
Array integers1( 7 );
```

invoca il costruttore di default (ricordate che il costruttore di default in questo esempio riceve un solo argomento `int` che ha valore di default 10). Questo costruttore (definito alla linea 47) convalida e assegna l'argomento al dato membro `size`, utilizza `new` per ottenere lo spazio che conterrà la rappresentazione interna dell'array e assegna il puntatore restituito da `new` al dato membro `ptr`, utilizza `assert` per verificare che l'operazione di `new` sia riuscita, incrementa `arrayCount` e infine utilizza un ciclo `for` per inizializzare tutti gli elementi dell'array a zero. È possibile creare una versione di `Array` che non inizializzi i suoi membri se, per esempio, devono essere letti in un secondo tempo. Ma non si tratta di una buona abitudine di programmazione. Gli `Array`, e tutti gli oggetti in genere, dovrebbero essere sempre in uno stato coerente e, dunque, dovrebbero essere inizializzati correttamente.

La linea 13

```
Array( const Array & );               // costruttore di copia
```

dichiara un *costruttore di copia* (definito alla linea 60) che inizializza un oggetto `Array` effettuando una copia da un altro oggetto `Array` esistente. Questa copia deve essere effettuata con attenzione, evitando che i due array puntino alla stessa area di memoria allocata dinamicamente, cosa che succederebbe se venisse effettuata la copia di default membro a membro.

I costruttori di copia sono invocati ogniqualvolta è necessario copiare un oggetto, come quando si passa un argomento per valore ad una funzione, quando si deve restituire un oggetto da una funzione o quando si inizializza un oggetto come copia di un altro oggetto della stessa classe. Il costruttore di copia viene chiamato nelle definizioni, quando si istanzia un oggetto `Array` inizializzandolo con un altro oggetto `Array`, come in:


```
Array integers3( integers1 );
```

o nella dichiarazione equivalente

```
Array integers3 = integers1;
```

Errore tipico 8.6

Ricordate che il costruttore di copia deve obbligatoriamente utilizzare la chiamata per riferimento altrimenti la chiamata al costruttore di copia comporterà una ricorsione infinita (un errore logico fatale). Infatti, utilizzando una chiamata per valore, al costruttore di copia verrebbe passata una copia dell'oggetto e per effettuare tale copia il compilatore invocherebbe ricorsivamente lo stesso costruttore di copia!

Il costruttore di copia di `Array` utilizza un inizializzatore di membro per copiare il membro `size` dell'array di inizializzazione nel membro `size` dell'altro oggetto, ottiene con `new` lo spazio in memoria che conterrà la rappresentazione interna dell'array e assegna il puntatore restituito da `new` a `ptr`, utilizza `assert` per verificare che l'operazione di `new` sia riuscita, incrementa `arrayCount` e infine utilizza un ciclo `for` per copiare, uno ad uno, tutti gli elementi dell'array di inizializzazione nel nuovo array.

Errore tipico 8.7

Se il costruttore di copia copiasse semplicemente il puntatore dell'oggetto origine nel puntatore dell'oggetto destinazione, entrambi gli oggetti punterebbero alla stessa area di memoria. Il distruttore eseguito per primo invaliderebbe quest'area, per cui il membro `ptr` dell'altro oggetto punterebbe a un'area indefinita: questa situazione prende il nome di "puntatore pendente" e generalmente causa seri errori durante l'esecuzione.

Ingegneria del software 8.4

Per ogni classe che fa uso dell'allocazione dinamica della memoria sono generalmente presenti in gruppo: un costruttore, un distruttore, l'overloading dell'operatore di assegnamento e un costruttore di copia.

La linea 14

```
~Array();                             // distruttore
```

dichiara il distruttore della classe (definito alla linea 71). Il distruttore viene invocato automaticamente quando il ciclo di vita di un oggetto `Array` giunge al termine. Esso utilizza `delete []` per restituire al sistema l'area allocata dinamicamente da `new` e quindi decrementa `arrayCount`.

La linea 15

```
int getSize() const;                  // restituisce size
```

dichiara una funzione che legge `size`.

La linea 16

```
const Array &operator=( const Array & ); // assegna gli array
```

dichiara l'overloading dell'operatore di assegnamento. Quando il compilatore incontra l'espressione

```
integers1 = integers2;
```

invoca la funzione `operator=` generando la chiamata

```
integers1.operator=( integers2 )
```

La funzione membro `operator=` (definita alla linea 82) verifica che non si tratti di un *autoassegnamento, nel qual caso l'assegnamento viene ignorato. Il motivo principale di* questo comportamento è che l'oggetto è già uguale a se stesso, ma vedremo che un autoassegnamento può essere persino pericoloso. Se non si tratta di autoassegnamento, la funzione membro determina se le dimensioni dei due array sono uguali: in questo caso l'array originario di interi dell'oggetto `Array` utilizzato come *lvalue* non viene riallocato. *Altrimenti, `operator=` utilizza `delete` per restituire al sistema lo spazio allocato originariamente,* copia `size` dell'array origine in `size` dell'array destinazione, utilizza `new` per allocare una nuova area di memoria delle dimensioni appropriate nell'array destinazione, pone il nuovo puntatore restituito da `new` in `ptr` e utilizza `assert` per verificare che l'operazione di `new` sia riuscita.

Successivamente `operator=` utilizza un ciclo `for` per copiare, uno alla volta, gli elementi dell'array origine in quelli corrispondenti dell'array destinazione. Al termine, la funzione restituisce l'oggetto corrente (`*this`) come riferimento costante: in questo modo sono possibili assegnamenti a cascata come `x = y = z`.

Errore tipico 8.8

Se una classe contiene puntatori ad aree di memoria allocate dinamicamente, dovete prevedere l'overloading dell'operatore di assegnamento e un costruttore di copia, altrimenti commetterete degli errori logici.

Ingegneria del software 8.5

È possibile inibire l'operazione di assegnamento tra oggetti di una data classe semplicemente dichiarando l'operatore di assegnamento come membro `private` della classe.

Ingegneria del software 8.6

È possibile inibire la copia di oggetti di una classe rendendo `private` sia l'overloading dell'operatore di assegnamento che il costruttore di copia.

La linea 17

```
bool operator==( const Array & ) const;  // confronto
```

dichiara l'overloading dell'operatore di uguaglianza (==). Quando il compilatore incontra l'espressione

```
integers1 == integers2
```

in `main`, invoca la funzione membro `operator==` generando la chiamata

```
integers1.operator==( integers2 )
```

La funzione membro `operator==` (definita alla linea 104) restituisce `false` immediatamente se i membri `size` dei due array sono diversi, altrimenti procede nel confronto di tutte le coppie di elementi. Se trova solo uguaglianze, restituisce `true`. Non appena trova una coppia di elementi non uguali, la funzione termina restituendo `false`.

Le linee 21 e 22

```
bool operator!=( const Array &right ) const
   { return ! ( *this == right ); }
```


definiscono l'overloading dell'operatore di disuguaglianza (`!=`). La funzione `operator!=` è definita in termini dell'operatore di uguaglianza: si limita infatti a chiamare la funzione `operator==` per determinare se i due `Array` sono uguali e a restituire il risultato complementare. Scrivere `operator!=` in termini di `operator==` abbrevia il codice e promuove il riutilizzo di una funzione già scritta. Inoltre osservate che la definizione completa di `operator!=` si trova nel file di intestazione di `Array`. Ciò consente al compilatore di rendere `inline` la definizione di `operator!=` e consente di migliorare l'efficienza avendo eliminato una chiamata di funzione.

Le linee 24 e 25

```
int &operator[]( int );              // operatore di indicizzazione
const int &operator[]( int ) const;  // operatore di indicizzazione
```

dichiarano due overloading dell'operatore di indicizzazione (definiti alle linee 118 e 128 rispettivamente). Quando il compilatore incontra l'espressione

```
integers1[ 5 ]
```

in `main`, invoca la funzione membro appropriata generando la chiamata

```
integers1.operator[]( 5 )
```

Il compilatore crea una chiamata alla versione `const` di `operator[]` quando l'operatore di indicizzazione viene utilizzato su un oggetto `Array` dichiarato come `const`. Ciascuna definizione di `operator[]` verifica la validità dell'indice, e se questo non è corretto, il programma termina. In caso contrario viene restituito l'elemento appropriato dell'array come riferimento, in modo che possa essere utilizzato come *lvalue* (ad esempio a sinistra di un operatore di assegnamento) se si tratta della versione non `const` di `operator[]`, o come *rvalue* se si tratta della versione `const`.

La linea 26

```
static int getArrayCount();          // restituisce count
```

dichiara la funzione `static getArrayCount` che restituisce il valore del dato membro `static arrayCount`, anche se non è stato istanziato alcun oggetto `Array`.

8.9 Conversioni tra tipi diversi

La maggior parte dei programmi elabora informazioni di vario tipo. A volte tutte le operazioni restano nell'ambito di un solo tipo di dato: per esempio l'addizione di due interi produce ancora un intero, sempre che il risultato non sia talmente grande da richiedere un *intero di dimensioni superiori. Tuttavia spesso si presenta la necessità di convertire dati di* un tipo in un altro tipo. Ciò accade negli assegnamenti, nei calcoli, nel passaggio di valori a una funzione e nella restituzione di un valore da una funzione. Il compilatore sa come effettuare le conversioni per i tipi predefiniti e, eventualmente, il programmatore può forzare le conversioni da un tipo predefinito a un altro tramite il cast.

Ma che cosa succede nel caso dei tipi definiti dall'utente? Il compilatore non può sapere come effettuare automaticamente la conversione tra tipi predefiniti e tipi definiti dall'utente ed è quindi il programmatore che deve specificare come effettuare ogni conversione. Le conversioni possono essere effettuate tramite i *costruttori di conversione*, che ricevono un solo argomento e che convertono gli oggetti di altri tipi (inclusi i tipi predefiniti) in oggetti di una determinata classe. In seguito faremo uso dei costruttori di conversione per convertire stringhe ordinarie (`char *`) in oggetti della classe `String`.

Per convertire un oggetto di una classe in oggetto di un'altra classe o di un tipo predefinito, si può utilizzare un *operatore di conversione* (detto anche *operatore di cast*): questo operatore deve essere una funzione membro non `static` e non è possibile definirlo come funzione `friend`.

Il prototipo di funzione

```
A::operator char *() const;
```

dichiara l'overloading dell'operatore di cast, per creare un oggetto temporaneo di tipo `char *` a partire da un oggetto del tipo `A` definito dall'utente. L'overloading della funzione operatore di cast non specifica il tipo di dato restituito: questo infatti corrisponde al tipo di dato in cui viene convertito l'oggetto. Se `s` è un oggetto di una classe, quando il compilatore incontra l'espressione `(char *) s` genera la chiamata `s.operator char *()`. L'operando `s` è l'oggetto per cui viene invocata la funzione membro operatore `operator char *`.

Gli operatori di cast possono convertire oggetti di un tipo definito dall'utente in tipi predefiniti o in altri tipi definiti dall'utente. I prototipi

```
A::operator int() const;
A::operator otherClass() const;
```

dichiarano overloading di funzioni operatore di cast, che convertono un oggetto del tipo definito dall'utente `A` in un intero e nel tipo definito dall'utente `otherClass`.

Una delle caratteristiche più apprezzate degli operatori di cast e dei costruttori di conversione è che, quando è necessario, il compilatore può chiamare automaticamente queste funzioni per creare oggetti temporanei. Per esempio, se in un programma compare un oggetto `s` della classe `String` al posto di un normalissimo `char *`, come in

```
cout << s;
```

il compilatore chiama la funzione `operator char *` per convertire l'oggetto in `char *` e utilizza il risultato di questo cast nell'espressione. Perciò non abbiamo neanche bisogno dell'overloading di `<<` per effettuare l'output di un oggetto `String` con `cout`.

8.10 Progettazione della classe String

Siamo arrivati al cuore di questo capitolo: progetteremo e implementeremo una classe che gestisce la creazione e la manipolazione di stringhe (Figura 8.5). La classe `string` oggi fa parte delle librerie standard del C++ e la studieremo in dettaglio nel Capitolo 7 del volume C++ Tecniche avanzate di programmazione. È un utile esercizio, comunque, scrivere una nostra versione della classe `String`.

Presenteremo prima l'intestazione della classe `String`, discutendo l'uso dei dati `private` che rappresentano gli oggetti `String`. Dopo di che esploreremo l'interfaccia pubblica della classe e i diversi servizi che offre. Successivamente vedremo come funziona la funzione `main` del programma di esempio. Discuteremo l'aspetto che dovrà avere il codice che manipola i nostri oggetti `String`, conciso e composto da espressioni basate soprattutto su operatori anziché su chiamate di funzione.

Infine passeremo in rassegna le varie funzioni membro della classe `String`. Per ogni operatore ridefinito, mostreremo il codice del programma di esempio che lo invoca e spiegheremo come funziona.


```
1   // Fig. 8.5: string1.h
2   // Definizione della classe String
3   #ifndef STRING1_H
4   #define STRING1_H
5
6   #include <iostream.h>
7
8   class String {
9      friend ostream &operator<<( ostream &, const String & );
10     friend istream &operator>>( istream &, String & );
11
12  public:
13     String( const char * = "" ); // costr. di conv./di default
14     String( const String & );    // costruttore di copia
15     ~String();                   // distruttore
16     const String &operator=( const String & );  // assegnamento
17     const String &operator+=( const String & ); // concatenamento
18     bool operator!() const; // l'ogg.String è la stringa vuota?
19     bool operator==( const String & ) const; // verifica se s1==s2
20     bool operator<( const String & ) const; // verifica se s1 < s2
21
22     // verifica se le due stringhe sono diverse
23     bool operator!=( const String & right ) const
24        { return !( *this == right ); }
25
26     // verifica se s1 è maggiore di s2
27     bool operator>( const String &right ) const
28        { return right < *this; }
29
30     // verifica se s1 è minore o uguale a s2
31     bool operator<=( const String &right ) const
32        { return !( right < *this ); }
33
34     // verifica se s1 è maggiore o uguale a s2
35     bool operator>=( const String &right ) const
36        { return !( *this < right ); }
37
38     char &operator[]( int );  // operatore di indicizzazione
39     const char &operator[]( int ) const; // operatore di ind.
40     String &operator()( int, int ); //restituisce una sottostringa
41     int getLength() const; //restituisce lunghezza della stringa
42
43  private:
44     int length;                    // lunghezza della stringa
45     char *sPtr;  // punta all'indirizzo di partenza della stringa
46
```

Figura 8.5 La classe String (continua)

```
47     void setString( const char * );  // funzione di utilità
48  };
49
50  #endif
51  // Fig. 8.5: string1.cpp
52  // Definizioni delle funzioni membro della classe String
53  #include <iostream.h>
54  #include <iomanip.h>
55  #include <string.h>
56  #include <assert.h>
57  #include "string1.h"
58
59  // Costruttore di conversione: converte  char * in String
60  String::String( const char *s ) : length( strlen( s ) )
61  {
62     cout << "Conversion constructor: " << s << '\n';
63     setString( s );          // chiama la funzione di utilità
64  }
65
66  // Costruttore di copia
67  String::String( const String &copy ) : length( copy.length )
68  {
69     cout << "Copy constructor: " << copy.sPtr << '\n';
70     setString( copy.sPtr ); // chiama la funzione di utilità
71  }
72
73  // Distruttore
74  String::~String()
75  {
76     cout << "Destructor: " << sPtr << '\n';
77     delete [] sPtr;//restituisce la memoria occupata dalla stringa
78  }
79
80  // Overloading di =; evita l'autoassegnamento
81  const String &String::operator=( const String &right )
82  {
83     cout << "operator= called\n";
84
85     if ( &right != this ) {          // evita l'autoassegnamento
86        delete [] sPtr;   // rilascia la memoria allocata in prec.
87        length = right.length;      // nuova lunghezza della String
88        setString( right.sPtr );    // chiama la funzione di utilità
89     }
90     else
91        cout << "Attempted assignment of a String to itself\n";
92
```

Figura 8.5 La classe String (continua)


```
93     return *this;  // consente di scrivere assegnamenti a cascata
94  }
95
96  // Concatena l'operando destro all'istanza corrente e vi
97  // memorizza il risultato.
98  const String &String::operator+=( const String &right )
99  {
100    char *tempPtr = sPtr; //memorizza per poter rilasciare la mem.
101    length += right.length;       // nuova lunghezza della String
102    sPtr = new char[ length + 1 ]; // crea spazio
103    assert( sPtr != 0 ); //termina se non è stata allocata memoria
104    strcpy( sPtr, tempPtr );  // parte sinistra della nuova String
105    strcat( sPtr, right.sPtr ); // parte destra della nuova String
106    delete [] tempPtr; // restituisce la memoria allocata in prec.
107    return *this;      // consente di scrivere chiamate a cascata
108 }
109
110 // L'oggetto String è la stringa vuota?
111 bool String::operator!() const { return length == 0; }
112
113 // Questa String è uguale a quella di destra?
114 bool String::operator==( const String &right ) const
115    { return strcmp( sPtr, right.sPtr ) == 0; }
116
117 // Questa String è minore di quella di destra?
118 bool String::operator<( const String &right ) const
119    { return strcmp( sPtr, right.sPtr ) < 0; }
120 // Restituisce un riferimento a un carattere in una String
121 // come lvalue.
122 char &String::operator[]( int subscript )
123 {
124    // Verifica prima che l'indice sia valido
125    assert( subscript >= 0 && subscript < length );
126
127    return sPtr[ subscript ];  // crea un  lvalue
128 }
129 // Restituisce un riferimento a un carattere in una String
130 // come rvalue.
131 const char &String::operator[]( int subscript ) const
132 {
133    // Verifica prima che l'indice sia valido
134    assert( subscript >= 0 && subscript < length );
135
136    return sPtr[ subscript ];  // crea un rvalue
137 }
138
```

Figura 8.5 La classe String (continua)

```
139 // Restituisce una sottostringa che parte da index e ha lunghezza
140 // pari a subLength come riferimento a un oggetto String.
141 String &String::operator()( int index, int subLength )
142 {                          // si assicura che index sia valido e che
143                            // la lunghezza della sottostringa >= 0
144    assert( index >= 0 && index < length && subLength >= 0 );
145
146    String *subPtr = new String; // String vuota
147    assert( subPtr != 0 ); // si assicura che sia stata allocata
148                           // la nuova String
149    // determina la lunghezza della sottostringa
150    if ( ( subLength == 0 ) || ( index + subLength > length ) )
151       subPtr->length = length - index + 1;
152    else
153       subPtr->length = subLength + 1;
154
155    // alloca nuova memoria per la sottostringa
156    delete subPtr->sPtr; // elimina l'array di caratteri dall'ogg.
157    subPtr->sPtr = new char[ subPtr->length ];
158    assert( subPtr->sPtr != 0 ); // si assicura che sia stato
159                                 // allocato nuovo spazio
160    // copia la sottostringa nella nuova String
161    strncpy( subPtr->sPtr, &sPtr[ index ], subPtr->length );
162    subPtr->sPtr[subPtr->length] = '\0'; //termina la nuova String
163
164    return *subPtr;        // restituisce la nuova String
165 }
166
167 // Restituisce la lunghezza della stringa
168 int String::getLength() const { return length; }
169
170 // Funzione di utilità che è chiamata dai costruttori e
171 // dall'operatore di assegnamento.
172 void String::setString( const char *string2 )
173 {
174    sPtr = new char[ length + 1 ]; // alloca memoria
175    assert( sPtr != 0 ); // termina se non è stata allocata mem.
176    strcpy( sPtr, string2 );      // copia il literal nell'oggetto
177 }
178
179 // Overloading dell'operatore di output
180 ostream &operator<<( ostream &output, const String &s )
181 {
182    output << s.sPtr;
183    return output;    // consente di scrivere istruzioni a cascata
184 }
185
186 // Overloading dell'operatore di input
```

Figura 8.5 La classe String (continua)


```
187 istream &operator>>( istream &input, String &s )
188 {
189    char temp[ 100 ];   // buffer per memorizzare l'input
190
191    input >> setw( 100 ) >> temp;
192    s = temp;  // utilizza l'operatore di ass. della classe String
193    return input;    // consente di scrivere istruzioni a cascata
194 }
195 // Fig. 8.5: fig08_05.cpp
196 // Programma di esempio per la classe String
197 #include <iostream.h>
198 #include "string1.h"
199
200 int main()
201 {
202    String s1( "happy" ), s2( " birthday" ), s3;
203
204 // verifica l'overloading degli op. relazionali e di uguaglianza
205    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
206         << "\"; s3 is \"" << s3 << '\"'
207         << "\nThe results of comparing s2 and s1:"
208         << "\ns2 == s1 yields "
209         << ( s2 == s1 ? "true" : "false" )
210         << "\ns2 != s1 yields "
211         << ( s2 != s1 ? "true" : "false" )
212         << "\ns2 >  s1 yields "
213         << ( s2 > s1 ? "true" : "false" )
214         << "\ns2 <  s1 yields "
215         << ( s2 < s1 ? "true" : "false" )
216         << "\ns2 >= s1 yields "
217         << ( s2 >= s1 ? "true" : "false" )
218         << "\ns2 <= s1 yields "
219         << ( s2 <= s1 ? "true" : "false" );
220
221    // verifica l'overloading dell'operatore "vuoto" (!)
222    cout << "\n\nTesting !s3:\n";
223    if ( !s3 ) {
224       cout << "s3 is empty; assigning s1 to s3:\n";
225       s3 = s1;       // verifica l'overloading dell'assegnamento
226       cout << "s3 is \"" << s3 << "\"";
227    }
228
229    // verifica l'overloading dell'operatore di concatenamento
230    cout << "\n\ns1 += s2 yields s1 = ";
231    s1 += s2; // verifica l'overloading dell'op. di concatenamento
232    cout << s1;
233
```

Figura 8.5 La classe String (continua)

```
234  // verifica il costruttore di conversione
235     cout << "\n\ns1 += \" to you\" yields\n";
236     s1 += " to you";      // verifica il costruttore di conversione
237     cout << "s1 = " << s1 << "\n\n";
238
239  // verifica l'overloading di () per l'estrazione di sottostringhe
240     cout << "The substring of s1 starting at\n"
241          << "location 0 for 14 characters, s1(0, 14), is:\n"
242          << s1( 0, 14 ) << "\n\n";
243  // verifica l'opzione di estrazione di sottostringhe
244  // "fino alla fine della String"
245     cout << "The substring of s1 starting at\n"
246          << "location 15, s1(15, 0), is: "
247          << s1( 15, 0 ) << "\n\n";  // 0 significa "fino alla fine
248                                     // della String "
249  // verifica il costruttore di copia
250     String *s4Ptr = new String(s1);
251     cout << "*s4Ptr = " << *s4Ptr << "\n\n";
252  // verifica il comportamento dell'operatore di assegnamento (=)
253  // in un autoassegnamento
254     cout << "assigning *s4Ptr to *s4Ptr\n";
255     *s4Ptr = *s4Ptr;      // verifica l'operatore di assegnamento
256     cout << "*s4Ptr = " << *s4Ptr << '\n';
257
258  // verifica il distruttore
259     delete s4Ptr;
260
261  // verifica la creazione di un lvalue con l'op. di indicizzazione
262     s1[ 0 ] = 'H';
263     s1[ 6 ] = 'B';
264     cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
265          << s1 << "\n\n";
266
267  // verifica un indice non valido
268     cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
269     s1[ 30 ] = 'd';       // ERRORE: indice non valido
270
271     return 0;
272  }
```

```
Conversion constructor: happy
Conversion constructor: birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
```

Figura 8.5 La classe String (continua)


```
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3:
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields:
Conversion constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Assertion failed: subscript >= 0 && subscript < length,
file String1.cpp, line 76

abnormal program termination
```

Figura 8.5 La classe String.

Cominciamo con la rappresentazione interna di un oggetto `String`. Le linee 44 e 45

```
int length;       // lunghezza della stringa
char *sPtr;       // puntatore all'indirizzo di partenza della stringa
```

dichiarano i dati membro `private` della classe. La nostra implementazione di `String` ha un campo `length` che indica il numero di caratteri della stringa, escluso il terminatore

nullo finale, e un puntatore `sPtr` all'area di memoria allocata dinamicamente che conterrà i caratteri effettivi della stringa.

Adesso facciamo un salto nel file di intestazione di `String` in Figura 8.5. Le linee 9 e 10

```
friend ostream &operator<<( ostream &, const String & );
friend istream &operator>>( istream &, String & );
```

dichiarano l'overloading dell'operatore di inserimento nello stream, la funzione `operator<<` (definita alla linea 180), e dell'operatore di estrazione dallo stream, la funzione `operator>>` (definita alla linea 187), entrambe come `friend` della classe. La loro implementazione è semplice e diretta.

Le linea 13

```
String( const char * = "" );     // costruttore di conversione -
                                  // di default
```

dichiara un *costruttore di conversione*. Questo costruttore (definito alla linea 60) riceve un argomento `const char *` (che per default è una stringa vuota) e istanzia un oggetto `String` che include la stessa stringa di caratteri. Ogni costruttore che riceve un solo argomento può essere visto come costruttore di conversione. Come vedremo, questi costruttori sono utili nelle manipolazioni di oggetti `String` in cui sono coinvolti argomenti `char *`. Il costruttore di conversione converte la stringa `char *` in un oggetto `String`, che viene quindi assegnato all'oggetto `String` di destinazione. Se c'è un costruttore di conversione, non è necessario scrivere anche l'overloading dell'operatore di assegnamento per assegnare stringhe ordinarie a oggetti `String`: è il compilatore stesso a invocare automaticamente il costruttore di conversione per creare un oggetto `String` temporaneo che contenga la stringa ordinaria. Successivamente viene invocato l'overloading dell'operatore di assegnamento per assegnare l'oggetto `String` temporaneo ad un altro oggetto `String`.

Ingegneria del software 8.7

Non è possibile effettuare catene di conversioni implicite: il linguaggio può applicare automaticamente un solo costruttore di conversione per cercare di soddisfare la segnatura di una funzione o di un operatore. Se ciò non è possibile viene generato un errore di sintassi.

Il costruttore di conversione di `String` può essere invocato in una dichiarazione del tipo `String s1( "happy" )`. Esso calcola la lunghezza della stringa di caratteri e la assegna al suo dato membro privato `length` nella lista di inizializzatori di membro, quindi chiama la funzione di utilità privata `setString`. La funzione `setString` (definita alla linea 172) utilizza `new` per fornire al dato membro privato `sPtr` un'area di memoria adeguata a cui puntare, utilizza `assert` per verificare che l'operazione di `new` sia stata eseguita con successo e, in caso affermativo, utilizza `strcpy` per copiare la stringa di caratteri nell'oggetto.

La linea 14

```
String( const String & );     // costruttore di copia
```

è un costruttore di copia (definito alla linea 67) che inizializza un oggetto `String` copiandone il contenuto da un altro oggetto `String` esistente. Questa copia deve essere effettuata con molta attenzione, evitando soprattutto di cadere nella trappola di far puntare entrambi gli oggetti `String` alla stessa area di memoria allocata dinamicamente (che è esattamente il

comportamento del costruttore di copia di default). Il costruttore di copia opera in modo simile al costruttore di conversione, eccetto per il fatto che copia semplicemente il membro `length` dall'oggetto `String` origine all'oggetto `String` destinazione. Osservate che il costruttore di copia crea nuovo spazio per la rappresentazione interna del nuovo oggetto `String`. Se infatti si limitasse a copiare il membro `sPtr` dell'oggetto origine nel membro `sPtr` dell'oggetto destinazione, entrambi gli oggetti finirebbero per puntare alla stessa area di memoria allocata dinamicamente. Dunque, il primo distruttore chiamato effettuerebbe una `delete` su quell'area dinamica, lasciando il puntatore `sPtr` dell'altro oggetto in uno stato indefinito (rendendolo cioè un *puntatore pendente*), una situazione che molto probabilmente causerebbe gravi errori durante l'esecuzione.

La linea 15

```
~String();                                  // distruttore
```

dichiara il distruttore della classe `String` (definito alla linea 74). Questo distruttore fa uso di `delete` per restituire al sistema la memoria allocata dinamicamente che conteneva la stringa di caratteri.

La linea 16

```
const String &operator=( const String & );  // assegnamento
```

dichiara l'overloading dell'operatore di assegnamento, la funzione `operator=` (definita alla linea 81). Quando il compilatore incontra un'espressione del tipo `string1 = string2`, genera la chiamata di funzione

```
string1.operator=( string2 );
```

La funzione `operator=` verifica se si tratti di un autoassegnamento e in questo caso restituisce semplicemente il controllo alla funzione chiamante. Se non ci fosse questo test, la funzione effettuerebbe una `delete` sullo spazio dell'oggetto destinazione, perdendo in questo modo la stringa di caratteri. Se la funzione non rileva un autoassegnamento, effettua l'operazione `delete` sullo spazio di memoria, copia il campo `length` dell'oggetto origine in quello dell'oggetto destinazione e chiama `setString` (linea 172) per creare un nuovo spazio per l'oggetto destinazione, determina se il compito di `new` è riuscito e utilizza `strcpy` per copiare la stringa di caratteri dell'oggetto origine nell'oggetto destinazione. In ogni caso, al termine della funzione viene restituito `*this` in modo che sia possibile scrivere assegnamenti a cascata.

La linea 17

```
const String &operator+=( const String & ); // concatenamento
```

dichiara l'overloading dell'operatore di concatenamento di stringhe (definito alla linea 98). Quando il compilatore incontra l'espressione `s1 += s2` in `main`, genera la chiamata `s1.operator+=( s2 )`. La funzione `operator+=` crea un puntatore temporaneo che contiene la stringa di caratteri dell'oggetto corrente in modo da poter effettuare una `delete` sulla memoria della stringa e calcola la lunghezza della combinazione delle due stringhe. Poi essa utilizza l'operatore `new` per riservare lo spazio necessario alla nuova stringa ed `assert` per verificare che l'operazione di `new` sia riuscita. La funzione di libreria `strcpy` è adoperata per copiare la stringa originaria nel nuovo spazio allocato ed è concatenata con quella dell'oggetto origine per mezzo della funzione `strcat`. In seguito, per restituire al sistema lo spazio occupato dalla stringa di caratteri contenuta precedentemente nell'oggetto corrente viene chiamato l'operatore `delete`. Al termine si restituisce `*this` (che è a tutti gli effetti un riferimento ad un oggetto `String`) per consentire di scrivere diversi operatori `+=` in cascata.

A questo punto la domanda è: abbiamo bisogno di un nuova ridefinizione per effettuare il concatenamento di un oggetto `String` e di un `char *`? La risposta è no. Il costruttore *di conversione per* `const char *` *converte una stringa convenzionale in un oggetto* `String` temporaneo, che può essere utilizzato dall'operatore di concatenamento appena descritto. Ricordiamo ancora una volta che il C++ non può effettuare conversioni in cascata per ottenere l'uguaglianza dei tipi richiesti dalle funzioni. Il C++ può al massimo effettuare una sola altra conversione implicita (definita dal compilatore), cioè tra tipi predefiniti, prima di procedere alla conversione tra un tipo predefinito e una classe. Osservate che quando viene creato un oggetto `String` temporaneo, sono invocati costruttore e distruttore di default (cfr. l'output di `s1 += " to you"` *in Figura 8.5*). *Queste chiamate di funzione hanno* un costo in termini di efficienza, ma è un aspetto che resta nascosto ai client della classe. Tale problema si verifica a causa dei costruttori di copia, quando si passano dei parametri a una funzione per valore o quando una funzione restituisce oggetti di una classe per valore.

Obiettivo efficienza 8.2

In termini di efficienza è preferibile il solo overloading dell'operatore di concatenamento `+=` *che riceve un solo argomento di tipo* `const char*`, *rispetto a una conversione implicita più un concatenamento. Le conversioni implicite abbreviano il codice e causano meno errori.*

La linea 18

```
bool operator!() const;                    // la stringa è vuota?
```

dichiara l'overloading dell'operatore di complementazione (definito alla linea 111). Per le classi stringa questo operatore verifica generalmente se una stringa è vuota. Per esempio, quando il compilatore incontra l'espressione `!string1`, genera la chiamata alla funzione

```
string1.operator!()
```

questa funzione restituisce semplicemente il risultato del test `length == 0`.

Le linee

```
bool operator==( const String & ) const;   // verifica se s1 == s2
bool operator<( const String & ) const;    // verifica se s1 < s2
```

dichiarano l'overloading dell'operatore di uguaglianza (definito alla linea 114) e dell'operatore "minore di" (definito alla linea 118). Tutti questi operatori sono simili, per cui discuteremo solo il caso dell'operatore `==`. Quando il compilatore incontra l'espressione `string1 == string2`, genera la chiamata alla funzione

```
string1.operator==( string2 )
```

che restituisce `true` *se* `string1` *è uguale a* `string2`. *Questi operatori chiamano la funzione* di libreria `strcmp` per confrontare le stringhe di caratteri degli oggetti `String`. Molti programmatori definiscono l'overloading degli operatori relazionali e di uguaglianza in termini di uno solo. Anche noi seguiamo questa convenzione e quindi implementiamo gli operatori `!=`, `>`, `<=` e `>=` in termini di `operator==` e `operator<` (linee 23-36). Per esempio, la funzione `operator>=` è implementata come segue alla linea 35 nel file di intestazione:

```
bool String::operator>=( const String &right ) const
   { return !( *this < right ); }
```


La funzione `operator>=` si serve dell'overloading di `<` per determinare se un oggetto `String` è maggiore o uguale a un altro. Notate che le funzioni per gli operatori `!=`, `>`, `<=` e `>=` *sono definite nel file di intestazione. Notate che queste funzioni sono implicitamente* `inline` e dunque non vi sono perdite di efficienza dovute ad un ulteriore invocazione di funzione.

Ingegneria del software 8.8

Implementando nuove funzioni membro in termini di funzioni già esistenti abbrevierete il vostro codice.

Le linee 38 e 39

```
char &operator[]( int );                   // operatore di indicizzazione
const char &operator[]( int ) const;       // operatore di indicizzazione
```

dichiarano la ridefinizione degli operatori di indicizzazione (definiti alle linee 122 e 131), uno per `String` non costanti e uno per `String` costanti. Quando il compilatore incontra un'espressione del tipo `string1[ 0 ]`, genera la chiamata `string1.operator[]( 0 )` (utilizzando la giusta versione di `operator[]`, a seconda che `String` sia un *lvalue* oppure un *rvalue*). La funzione `operator[]` contiene prima una `assert` che controlla la validità dell'indice: se l'indice non è accettabile viene visualizzato un messaggio di errore e il programma termina immediatamente. Se invece l'indice è valido essa restituisce un riferimento (più precisamente un dato di tipo `char &`) al carattere desiderato dell'oggetto `String`: questo riferimento può essere utilizzato come *lvalue* per modificare tale carattere nell'oggetto `String`. La versione `const` di `operator[]` restituisce un riferimento costante al carattere desiderato (`const char &`) dell'oggetto `String`; questo riferimento può essere utilizzato come *rvalue* per leggere il valore di tale carattere.

Collaudo e messa a punto 8.1

Attenzione: può essere pericoloso restituire un riferimento non costante nella ridefinizione di un operatore di indicizzazione. Il client infatti potrebbe utilizzarlo per modificare impropriamente il dato (ad esempio, nel caso delle stringhe inserendo un terminatore nullo `('\0')` *in qualsiasi punto della stringa).*

La linea 40

```
String &operator()( int, int ); // restituisce una sottotringa
```

dichiara l'overloading dell'operatore di chiamata di funzione (definito alla linea 141). Nelle classi stringa l'overloading di questo operatore serve per selezionare una sottostringa da un oggetto `String`. I due parametri interi specificano l'indirizzo di partenza e la lunghezza della sottostringa da estrarre. Se l'indirizzo di partenza non è corretto o la lunghezza è negativa, viene generato un messaggio di errore. Per convenzione, se la lunghezza della sottostringa è 0, viene restituita la sottostringa che inizia dall'indirizzo di partenza e termina alla fine dell'oggetto `String`. Per esempio, supponiamo che `string1` sia un oggetto `String` contenente la stringa di caratteri `"AEIOU"`. Quando il compilatore incontra l'espressione `string1( 2, 2 )`, genera la chiamata `string1.operator()( 2, 2 )` che produce un nuovo oggetto `String` allocato dinamicamente contenente la stringa `"IO"`.

L'overloading dell'operatore di chiamata di funzione `()` ha delle enormi potenzialità, perché le chiamate di funzione possono ricevere liste di parametri di lunghezza e complessità arbitraria. Ad esempio, si potrebbe utilizzare l'operatore `()` come sostituto dell'operatore di indicizzazione: anziché utilizzare la scomoda notazione del C `a[ b ][ c ]` per gli

array bidimensionali, alcuni programmatori preferiscono effettuare l'overloading di `()` e scrivere la stessa espressione nella forma `a( b, c )`. L'operatore di chiamata di funzione dev'essere obbligatoriamente ridefinito come funzione membro non `static`. Questo operatore viene invocato quando il "nome della funzione" è un oggetto della classe `String` come in

```
s1(0,14)
```

(si veda la linea 242).

La linea 41

```
int getLength() const;          // restituisce length
```

dichiara una funzione che restituisce la lunghezza dell'oggetto `String`. Osservate che questa funzione (definita alla linea 168) accede al dato membro privato `length` di `String`.

A questo punto dovreste essere in grado di comprendere autonomamente il codice di `main`, di esaminare l'output e di studiare come viene utilizzato ciascun operatore.

8.11 L'overloading degli operatori ++ e --

Anche gli operatori di incremento e decremento (preincremento, postincremento, predecremento e postdecremento) possono essere ridefiniti. Vedremo tra breve come il compilatore distingue tra le versioni prefisse e suffisse.

Per effettuare l'overloading dell'operatore di incremento in entrambe le versioni ogni funzione deve avere una segnatura diversa, in modo che il compilatore possa determinare quale versione di ++ si desidera ridefinire. Per l'overloading delle versioni prefisse si procede esattamente nello stesso modo di tutti gli altri operatori unari.

Supponiamo, per esempio di voler incrementare di 1 il membro `day` in un oggetto `Date` di nome `d1`. Quando il compilatore incontra questa espressione di preincremento

```
++d1
```

genera la chiamata alla funzione

```
d1.operator++()
```

il cui prototipo sarà

```
Date &operator++();
```

Se il preincremento è implementato come funzione non membro, quando il compilatore incontra l'espressione

```
++d1
```

genera la chiamata alla funzione

```
operator++( d1 )
```

il cui prototipo in `Date` sarà

```
friend Date &operator++( Date & );
```

L'overloading dell'operatore di postincremento invece è leggermente più problematico, perché bisogna consentire al compilatore di distinguere la funzione da quella di preincremento. La convenzione adottata in C++ è che quando il compilatore incontra un'espressione di postincremento come

```
d1++
```


genera la chiamata alla funzione

```
d1.operator++( 0 )
```

il cui prototipo sarà

```
Date operator++( int )
```

Lo `0` è in realtà soltanto un valore fittizio, serve soltanto a creare una lista di argomenti, in modo che la lista di argomenti di `operator++` per il postincremento sia differente da quella di `operator++` per il preincremento.

Se il postincremento è implementato come funzione non membro, quando il compilatore incontra l'espressione

```
d1++
```

genera la chiamata alla funzione

```
operator++(d1, 0)
```

il cui prototipo sarà

```
friend Date operator++( Date &, int );
```

Anche in questo caso lo `0` è un argomento fittizio, che serve a diversificare la lista di argomenti di `operator++` per il postincremento da quella per il preincremento.

Tutto questo vale allo stesso modo per gli operatori di predecremento e postdecremento. Nella prossima sezione esamineremo una versione della classe `Date` (che vi era stato chiesto di definire nell'esercizio 7.7) che fa uso dell'overloading degli operatori di preincremento e postincremento.

8.12 Progettazione della classe Date

La Figura 8.6 illustra la classe `Date`, il cui compito è rappresentare le date. L'implementazione di questa classe fa uso dell'overloading degli operatori di preincremento e postincremento per incrementare di 1 il membro `day` (giorno) in un oggetto `Date`, regolando di conseguenza anche `month` (mese) e `year` (anno), se necessario.

```
1  // Fig. 8.6: date1.h
2  // Definizione della classe Date
3  #ifndef DATE1_H
4  #define DATE1_H
5  #include <iostream.h>
6
7  class Date {
8     friend ostream &operator<<( ostream &, const Date & );
9
10 public:
11    Date( int m = 1, int d = 1, int y = 1900 ); // costruttore
12    void setDate( int, int, int ); // imposta la data
13    Date &operator++();            // operatore di preincremento
14    Date operator++( int );        // operatore di postincremento
```

Figura 8.6 La classe Date, che fa uso degli operatori di incremento in overloading (continua)

```
15     const Date &operator+=( int ); // aggiunge i giorni
16     bool leapYear( int );          // anno bisestile?
17     int endOfMonth( int );         // fine del mese?
18
19  private:
20     int month;
21     int day;
22     int year;
23
24     static const int days[];       // array di giorni per ogni mese
25     void helpIncrement();          // funzione di utilità
26  };
27
28  #endif
```

```
29  // Fig. 8.6: date1.cpp
30  // Definizioni delle funzioni membro per la classe Date
31  #include <iostream.h>
32  #include "date1.h"
33
34  // Inizializza i membri static nello scope a livello di file;
35  // una sola copia valida per tutti gli oggetti della classe.
36  const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
37                             31, 31, 30, 31, 30, 31 };
38
39  // costruttore di Date
40  Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
41
42  // Imposta la data
43  void Date::setDate( int mm, int dd, int yy )
44  {
45     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
46     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
47
48     // verifica se è un anno bisestile
49     if ( month == 2 && leapYear( year ) )
50        day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
51     else
52        day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
53  }
54
55  // Overloading dell'op. di preincremento come funzione membro.
56  Date &Date::operator++()
57  {
58     helpIncrement();
59     return *this;  // restituisce un rif. per creare un lvalue
60  }
```

Figura 8.6 La classe Date, che fa uso degli operatori di incremento in overloading (continua)


```
61
62  // Overloading dell'op. di postincremento come funzione membro.
63  // Notate che il parametro intero fittizio non ha
64  // un nome.
65  Date Date::operator++( int )
66  {
67     Date temp = *this;
68     helpIncrement();
69
70     // restituisce l'oggetto temporaneo salvato non incrementato
71     return temp;   // restituisce un valore e non un riferimento
72  }
73
74  // Aggiunge un numero specifico di giorni alla data
75  const Date &Date::operator+=( int additionalDays )
76  {
77     for ( int i = 0; i < additionalDays; i++ )
78        helpIncrement();
79
80     return *this;    // consente di scrivere istruzioni a cascata
81  }
82
83  // Se l'anno è bisestile restituisce true;
84  // altrimenti false
85  bool Date::leapYear( int y )
86  {
87     if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
88        return true;   // anno bisestile
89     else
90        return false;  // anno non bisestile
91  }
92
```

Figura 8.6 La classe Date, che fa uso degli operatori di incremento in overloading (continua)

L'interfaccia pubblica di `Date` include l'overloading dell'operatore di inserimento nello stream, un costruttore di default, la funzione `setDate`, l'overloading dell'operatore di preincremento, l'overloading dell'operatore di postincremento, l'overloading dell'operatore di assegnamento additivo (`+=`), una funzione per verificare se l'anno in corso è bisestile e una funzione per determinare se `day` è l'ultimo giorno del mese.

```
93  // Determina se il giorno è l'ultimo giorno del mese
94  bool Date::endOfMonth( int d )
95  {
96     if ( month == 2 && leapYear( year ) )
97        return d == 29; // ultimo giorno di Feb. in un anno bis.
98     else
```

Figura 8.6 La classe Date, che fa uso degli operatori di incremento in overloading (continua)

```
 99        return d == days[ month ];
100  }
101
102  // Funzione di servizio per incrementare la data
103  void Date::helpIncrement()
104  {
105     if ( endOfMonth( day ) && month == 12 ) {  // fine dell'anno
106        day = 1;
107        month = 1;
108        ++year;
109     }
110     else if ( endOfMonth( day ) ) {            // fine del mese
111        day = 1;
112        ++month;
113     }
114     else     // né fine del mese né fine dell'anno; incrementa day
115        ++day;
116  }
117
118  // Overloading dell'operatore di output
119  ostream &operator<<( ostream &output, const Date &d )
120  {
121     static char *monthName[ 13 ] = { "", "January",
122        "February", "March", "April", "May", "June",
123        "July", "August", "September", "October",
124        "November", "December" };
125
126     output << monthName[ d.month ] << ' '
127            << d.day << ", " << d.year;
128
129     return output;   // consente di scrivere istruzioni a cascata
130  }

131  // Fig. 8.6: fig08_06.cpp
132  // Programma di esempio per la classe Date
133  #include <iostream.h>
134  #include "date1.h"
135
136  int main()
137  {
138     Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
139     cout << "d1 is " << d1
140          << "\nd2 is " << d2
141          << "\nd3 is " << d3 << "\n\n";
142
143     cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
144
145     d3.setDate( 2, 28, 1992 );
```

Figura 8.6 La classe Date, che fa uso degli operatori di incremento in overloading (continua)


```
146     cout << "  d3 is " << d3;
147     cout << "\n++d3 is " << ++d3 << "\n\n";
148
149     Date d4( 3, 18, 1969 );
150
151     cout << "Testing the preincrement operator:\n"
152          << "  d4 is " << d4 << '\n';
153     cout << "++d4 is " << ++d4 << '\n';
154     cout << "  d4 is " << d4 << "\n\n";
155
156     cout << "Testing the postincrement operator:\n"
157          << "  d4 is " << d4 << '\n';
158     cout << "d4++ is " << d4++ << '\n';
159     cout << "  d4 is " << d4 << endl;
160
161     return 0;
162  }
```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Figura 8.6 La classe Date, che fa uso degli operatori di incremento in overloading.

La funzione `main` del programma di esempio crea l'oggetto `d1` inizializzandolo per default a `January 1, 1900`, l'oggetto `d2` inizializzandolo a `December 27, 1992` e l'oggetto `d3`, che il programma tenta di inizializzare a una data non valida. Il costruttore di `Date` chiama `setDate` per convalidare i valori specificati per `month`, `day` e `year`. Se `month` non è valido viene impostato a 1. Un valore di `year` non valido viene sostituito con 1900, mentre un valore di `day` non valido con 1.

Il programma di esempio esegue l'output di ogni oggetto `Date` tramite l'operatore di inserimento. L'overloading di `+=`, poi, aggiunge 7 giorni a `d2` e, quindi, la funzione `setDate` imposta `d3` a `February 28, 1992`. Successivamente viene creato un nuovo oggetto `Date` di nome `d4`, che viene impostato a `March 18, 1969`. Esso viene poi incrementato di 1

sfruttando la ridefinizione dell'operatore di preincremento. La data viene visualizzata due volte, prima e dopo il preincremento, per confermare che l'operazione è riuscita. Infine, `d4` è incrementato tramite l'overloading dell'operatore di postincremento ed anche in questo caso la data viene visualizzata prima e dopo l'operazione per verificarne la correttezza.

L'overloading dell'operatore di preincremento è semplice e diretto: esso chiama la funzione di utilità privata `helpIncrement` per incrementare la data. Questa funzione si occupa di gestire i cambi di mese (e di anno) che si verificano quando viene raggiunto l'ultimo giorno di un mese (o dell'anno). La funzione `helpIncrement` si serve delle funzioni `leapYear` (anno bisestile) ed `endOfMonth` (fine del mese) per incrementare correttamente `day`.

L'operatore di preincremento restituisce un riferimento all'oggetto `Date` corrente, ovvero quello che è stato appena incrementato.

L'overloading dell'operatore di postincremento è un po' più macchinoso; infatti, per simulare l'effetto di un postincremento, occorre restituire una copia non incrementata dell'oggetto `Date`. All'entrata della funzione `operator++`, salviamo quindi la copia corrente dell'oggetto (`*this`) in `temp`. Quindi chiamiamo `helpIncrement` per incrementare l'oggetto `Date` corrente. Infine restituiamo la copia non incrementata che abbiamo salvato. Osservate che questa funzione non può restituire un riferimento all'oggetto `Date` locale `temp`, perché le variabili locali sono distrutte all'uscita della funzione in cui sono dichiarate. Perciò se dichiarassimo il tipo restituito da questa funzione come `Date &` avremmo un riferimento a un oggetto che non esiste più. Questo è un errore tipico e alcuni compilatori sono in grado di rilevarlo visualizzando un avvertimento.

Esercizi di autovalutazione

8.1 Completate le seguenti affermazioni:

a) Supponiamo che `a` e `b` siano variabili intere e che `c` e `d` siano variabili a virgola mobile. Consideriamo le due somme `a + b` e `c + d`. I due operatori `+` sono stati utilizzati per operazioni chiaramente differenti. Questo è un esempio di .

b) La parola chiave _____ introduce la definizione di una funzione operatore in overloading.

c) Per utilizzare degli operatori sugli oggetti di una classe, occorre effettuarne l'overloading, tranne che per gli operatori _____ e _____.

d) L'overloading di un operatore non può mutare la _____, la _____ e il _____ di un operatore.

8.2 Spiegate i diversi significati degli operatori `<<` e `>>` in C++.

8.3 In quale contesto potreste trovare utilizzato il nome `operator/` in C++?

8.4 (Vera/Falsa) In C++ si può effettuare l'overloading soltanto degli operatori già esistenti.

8.5 Che differenza c'è tra la precedenza di un operatore in overloading e quella dell'operatore originario?

Risposte agli esercizi di autovalutazione

8.1 a) overloading di un operatore. b) `operator`. c) di assegnamento (`=`), indirizzo (`&`). d) precedenza, associatività, numero di operandi.

8.2 L'operatore `>>` effettua lo shifting dei bit a destra o l'estrazione da uno stream di input, a seconda del contesto in cui è utilizzato. L'operatore `<<` effettua lo shifting dei bit a sinistra o l'inserimento in uno stream di output, a seconda del contesto in cui è utilizzato.

8.3 Per l'overloading di un operatore: sarebbe il nome di una funzione che effettua l'overloading dell'operatore `/`.

8.4 Vera.

8.5 Nessuna.

Esercizi

8.6 Fate quanti più esempi potete di overloading implicito degli operatori del C++. Indicate un esempio significativo di situazione in cui desidereresti effettuare un overloading esplicito.

8.7 Gli operatori del C++ che non possono essere ridefiniti sono _____, _____, _____ e _____.

8.8 Il concatenamento di stringhe richiede due operandi, ovvero le due stringhe da concatenare. Nel testo vi abbiamo mostrato come implementare un operatore di concatenamento che concatena l'oggetto `String` sul lato destro dell'operatore a quello sul lato sinistro modificandolo. In alcune applicazioni si può desiderare di produrre un nuovo oggetto `String` risultante dal concatenamento di due oggetti `String` già esistenti, senza che questi siano modificati. A questo scopo implementate l'operatore di addizione per poter effettuare operazioni come la seguente

```
string1 = string2 + string3;
```

8.9 (*Stabilite tutti gli overloading possibili*) Per capire l'attenzione con cui bisogna scegliere gli operatori di cui effettuare l'overloading, elencate tutti gli operatori del C++ che possono essere ridefiniti, e per ognuno di essi elencate tutti i possibili significati che possono assumere, in corrispondenza con le classi che abbiamo incontrato finora. Vi suggeriamo di iniziare da:

a) Array
b) Pile
c) Stringhe

Dopo aver fatto ciò, indicate quali operatori sembrano avere significato per il maggior numero di classi, quali per il minor numero di classi e quali sembrano avere significato ambiguo.

8.10 *Adesso svolgete l'esercizio precedente al contrario. Elencate tutti gli operatori del C++ che* possono essere ridefiniti. Per ognuno di essi stabilite quale, secondo voi, potrebbe essere l'operazione migliore che dovrebbe svolgere su una data classe: se ci sono diverse alternative tutte ottime, elencatele tutte.

8.11 (*Diventate progettisti*) Quali operatori aggiungereste al C++ o a un futuro linguaggio simile al C++, che supportasse cioè sia la programmazione procedurale che quella orientata agli oggetti? Motivate la vostra scelta.

8.12 *Un esempio apprezzabile di overloading dell'operatore di chiamata di funzione* `()` *è facilitare* l'indicizzazione di array multidimensionali. Invece di dover scrivere

```
chessBoard[ row ][ column ]
```

per un array di oggetti, effettuate l'overloading di `()` per poter scrivere

```
chessBoard( row, column )
```

8.13 Create la classe `DoubleSubscriptedArray` (array bidimensionali) con caratteristiche simili alla classe in `Array` di Figura 8.4. Durante la fase di costruzione la classe dovrebbe poter creare un array di qualsiasi numero di righe e colonne. La classe dovrebbe prevedere la funzione `operator()` per

l'indicizzazione semplice degli elementi. Per esempio, per un oggetto `DoubleSubscriptedArray` 3x5 di nome a, l'utente deve poter scrivere `a( 1, 3 )` per accedere all'elemento che si trova nella prima riga e nella terza colonna. Ricordate che `operator()` può ricevere un numero qualsiasi di argomenti (cfr. la classe `String` in Figura 18.5 per un esempio di `operator()`). La rappresentazione dell'array bidimensionale è un array monodimensionale di interi con un numero di elementi pari a `righe x colonne`. La funzione `operator()` gestirà l'aritmetica dei puntatori in modo appropriato per accedere a ciascun elemento dell'array. Dovrebbero esserci due versioni di `operator()`, una che restituisce `int &` in modo che un elemento di un `DoubleSubscriptedArray` possa essere utilizzato come lvalue e un'altra che restituisce `const int &` in modo che un elemento di un `const DoubleSubscriptedArray` possa essere utilizzato come rvalue. Scrivete anche l'overloading dei seguenti operatori: `==`, `!=`, `=`, `<<` (per l'output dell'array in formato tabulare) e `>>` (per l'input di interi array).

8.14 Effettuate l'overloading dell'operatore di indicizzazione, in modo che restituisca l'elemento più grande di una collezione, poi il secondo in ordine di grandezza, poi il terzo e così via.

8.15 Considerate la classe `Complex` in Figura 8.7. Questa classe consente di effettuare operazioni sui numeri complessi, che possono essere pensati come costituiti da una coppia di interi: `realPart` (parte reale) e `imaginaryPart` (parte immaginaria) e rappresentati come `realPart + i* imaginaryPart` dove `i` è la radice quadrata di –1.

a) Modificate la classe, in modo che sia possibile l'input e l'output di numeri complessi tramite gli operatori `>>` e `<<` rispettivamente (eliminate dalla classe la funzione `print`).
b) Effettuate l'overloading dell'operatore di moltiplicazione, per effettuare la moltiplicazione di due numeri complessi secondo la seguente regola: (x + i*y)*(z + i*w) = (x*z – y*w) + i*(y*z + x*w)
c) Effettuate l'overloading degli operatori `==` e `!=` per poter mettere a confronto due numeri complessi.

```
1  // Fig. 8.7: complex1.h
2  // Definizione della classe Complex
3  #ifndef COMPLEX1_H
4  #define COMPLEX1_H
5
6  class Complex {
7  public:
8     Complex( double = 0.0, double = 0.0 );        // costruttore
9     Complex operator+( const Complex & ) const;   // addizione
10    Complex operator-( const Complex & ) const;   // sottrazione
11    const Complex &operator=( const Complex & ); // assegnamento
12    void print() const;                           // output
13 private:
14    double real;       // parte reale
15    double imaginary;  // parte immaginaria
16 };
17
18 #endif
19 // Fig. 8.7: complex1.cpp
20 // Definizioni delle funzioni membro della classe Complex
21 #include <iostream.h>
22 #include "complex1.h"
23
24 // costruttore
25 Complex::Complex( double r, double i )
26    : real( r ), imaginary( i ) { }
27
```

Figura 8.7 La classe Complex (continua)


```
28 // Overloading dell'operatore di addizione
29 Complex Complex::operator+( const Complex &operand2 ) const
30 {
31    return Complex( real + operand2.real,
32                    imaginary + operand2.imaginary );
33 }
34
35 // Overloading dell'operatore di sottrazione
36 Complex Complex::operator-( const Complex &operand2 ) const
37 {
38    return Complex( real   operand2.real,
39                    imaginary - operand2.imaginary );
40 }
41
42 // Overloading di =
43 const Complex& Complex::operator=( const Complex &right )
44 {
45    real = right.real;
46    imaginary = right.imaginary;
47    return *this;   // enables cascading
48 }
49
50 // Visualizza un Complex nella forma: (a, b)
51 void Complex::print() const
52    { cout << '(' << real << ", " << imaginary << ')'; }
53 // Fig. 8.7: fig08_07.cpp
54 // Programma di esempio per la classe Complex
55 #include <iostream.h>
56 #include "complex1.h"
57
58 int main()
59 {
60    Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );
61
62    cout << "x: ";
63    x.print();
64    cout << "\ny: ";
65    y.print();
66    cout << "\nz: ";
67    z.print();
68
69    x = y + z;
70    cout << "\n\nx = y + z:\n";
71    x.print();
72    cout << " = ";
73    y.print();
74    cout << " + ";
75    z.print();
76
77    x = y - z;
78    cout << "\n\nx = y   z:\n";
79    x.print();
80    cout << " = ";
81    y.print();
```

Figura 8.7 La classe Complex (continua)

```
82        cout << " - ";
83        z.print();
84        cout << endl;
85
86        return 0;
87     }
```

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Figura 8.7 La classe Complex.

8.16 Una macchina con interi rappresentati a 32 bit può manipolare gli interi che si trovano approssimativamente in un intervallo che va da –2 miliardi a +2 miliardi, che è sufficiente nella quasi totalità delle possibili applicazioni. Tuttavia ci sono applicazioni in cui sono necessari interi di più grandi dimensione, e ciò può essere realizzato per mezzo della classe `HugeInt` presentata in Figura 8.8. Studiate per bene la sua definizione e poi:

a) Descrivete in modo preciso come funziona.
b) Dite che restrizioni impone?
c) Effettuate l'overloading dell'operatore di moltiplicazione *
d) Effettuate l'overloading dell'operatore di divisione /.
e) Effettuate l'overloading di tutti gli operatori relazionali e di uguaglianza.

```
1     // Fig. 8.8: hugeint1.h
2     // Definizione della classe HugeInt
3     #ifndef HUGEINT1_H
4     #define HUGEINT1_H
5
6     #include <iostream.h>
7
8     class HugeInt {
9        friend ostream &operator<<( ostream &, HugeInt & );
10    public:
11       HugeInt( long = 0 );       // costruttore di conversione / di default
12       HugeInt( const char * );   // costruttore di conversione
13       HugeInt operator+( HugeInt & );    // somma con un altro HugeInt
14       HugeInt operator+( int );          // somma con un int
15       HugeInt operator+( const char * ); // somma un int con un char *
16    private:
17       short integer[30];
18    };
19
20    #endif
21    // Fig. 8.8: hugeint1.cpp
22    // Definizioni delle funzioni membro e friend della classe HugeInt
23    #include <string.h>
```

Figura 8.8 La classe HugeInt (continua)


```
24    #include "hugeint1.h"
25
26    // costruttore di conversione
27    HugeInt::HugeInt( long val )
28    {
29       int i;
30
31       for ( i = 0; i <= 29; i++ )
32          integer[ i ] = 0;    // inizializza l'array a zero
33
34       for ( i = 29; val != 0 && i >= 0; i-- ) {
35          integer[ i ] = val % 10;
36          val /= 10;
37       }
38    }
39
40    HugeInt::HugeInt( const char *string )
41    {
42       int i, j;
43
44       for ( i = 0; i <= 29; i++ )
45          integer[ i ] = 0;
46
47       for ( i = 30 - strlen( string ), j = 0; i <= 29; i++, j++ )
48          integer[ i ] = string[ j ] - '0';
49    }
50
51    // Addizione
52    HugeInt HugeInt::operator+( HugeInt &op2 )
53    {
54       HugeInt temp;
55       int carry = 0;
56
57       for ( int i = 29; i >= 0; i-- ) {
58          temp.integer[ i ] = integer[ i ] +
59                              op2.integer[ i ] + carry;
60
61          if ( temp.integer[ i ] > 9 ) {
62             temp.integer[ i ] %= 10;
63             carry = 1;
64          }
65          else
66             carry = 0;
67       }
68
69       return temp;
70    }
71
72    // Addizione
73    HugeInt HugeInt::operator+( int op2 )
74       { return *this + HugeInt( op2 ); }
75
76    // Addizione
77    HugeInt HugeInt::operator+( const char *op2 )
78       { return *this + HugeInt( op2 ); }
```

Figura 8.8 La classe HugeInt (continua)

```
79
80  ostream& operator<<( ostream &output, HugeInt &num )
81  {
82     int i;
83
84     for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
85        ; // skip leading zeros
86
87     if ( i == 30 )
88        output << 0;
89     else
90        for ( ; i <= 29; i++ )
91           output << num.integer[ i ];
92
93     return output;
94  }
95  // Fig. 8.8: fig08_08.cpp
96  // Programma di esempio per la classe HugeInt
97  #include <iostream.h>
98  #include "hugeint1.h"
99
100 int main()
101 {
102    HugeInt n1( 7654321 ), n2( 7891234 ),
103            n3( "99999999999999999999999999999999" ),
104            n4( "1" ), n5;
105
106    cout << "n1 is " << n1 << "\nn2 is " << n2
107         << "\nn3 is " << n3 << "\nn4 is " << n4
108         << "\nn5 is " << n5 << "\n\n";
109
110    n5 = n1 + n2;
111    cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
112
113    cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 )
114         << "\n\n";
115
116    n5 = n1 + 9;
117    cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
118
119    n5 = n2 + "10000";
120    cout << n2 << " + " << "10000" << " = " << n5 << endl;
121
122    return 0;
123 }
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999999 + 1
= 100000000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Figura 8.8 La classe HugeInt.

8.17 Create la classe `RationalNumber` (numeri razionali) con le seguenti caratteristiche:

a) Create un costruttore che inibisce il caso di uno 0 al denominatore, memorizza le frazioni in forma semplificata (ad es. la forma semplificata di 7/14 è 1/2), compiendo l'operazione di semplificazione qualora ciò fosse necessario, ed evita i numeri negativi al denominatore.
b) Effettuate l'overloading degli operatori di addizione, sottrazione, moltiplicazione e divisione per questa classe.
c) Effettuate l'overloading degli operatori relazionali e di uguaglianza per questa classe.

8.18 Studiate le funzioni di libreria del C per il trattamento delle stringhe e implementate ciascuna di esse nella classe `String`. Utilizzatele per quest'ultime per effettuare manipolazioni sui testi.

8.19 Un polinomio in un incognita x è la somma di monomi in x. Un monomio in x è costituito da un coefficiente moltiplicativo e dalla variabile x eventualmente elevata ad un esponente intero detto *grado*. Un esempio di polinomio è il seguente: 2*x^4+3*x^2-2*x+x+1. Progettate e scrivete la classe `Polynomial` che rappresenti internamente un polinomio come un array di termini, ciascuno dei quali possiede un coefficiente e un esponente. Ad esempio, il termine

2*x^4

ha coefficiente 2 ed esponente 4. La classe dovrebbe contenere costruttori e distruttori adeguati, così come funzioni *set* e *get*. Inoltre dovrebbe ridefinire anche i seguenti operatori:

a) Overloading dell'operatore di addizione (+) per sommare due `Polynomial` (la somma viene fatta termine a termine sui monomi aventi lo stesso grado: ad es. (2*x^2 - 2*x + 1) + (x^3 + 2*x) = x^3 + 2*x^2 + (-2+2)*x + 1 = x^3 + 2*x^2 + 1).
b) Overloading dell'operatore di sottrazione (-) per sottrarre un `Polynomial` da un altro (come prima, cambiando di segno i coefficienti del secondo polinomio).
c) Overloading dell'operatore di assegnamento per assegnare un `Polynomial` a un altro.
d) Overloading dell'operatore di moltiplicazione (*) per moltiplicare due `Polynomial` (il prodotto viene effettuato termine a termine moltiplicando i coefficienti e sommando gli esponenti risultanti: ad es. (2*x^2 – 2*x + 1) * (x^3 + 2*x) = 2*x^2*x^3 – 2*x*x^3 + 1*x^3 + 2*x^3*2*x –2*x*2*x + 1*2*x = 2*x^5 – 2*x^4 + x^3 + 4*x^4 – 4*x^2 + 2*x = 2*x^5 + 2*x^4 + x^3 – 4*x^2 + 2*x)
e) *Overloading dell'operatore di assegnamento con addizione (+=), sottrazione* (-=) e moltiplicazione (*=).

8.20 Il programma in Figura 8.3 contiene il commento

```
// Overloading dell'operatore di inserimento nello stream (non può
// essere funzione membro se si vuole invocarlo nella forma
// cout << somePhoneNumber)
```

In realtà non può essere funzione membro della classe `ostream`, ma può esserlo della classe `PhoneNumber` se accettiamo di doverlo invocare in questi modi:

```
somePhoneNumber.operator<<( cout );
```

o

```
somePhoneNumber << cout;
```

Riscrivete il programma in Figura 8.3 con `operator<<` come funzione membro, e riscrivete le due istruzioni precedenti per accertarvi che funzionino.

CAPITOLO 9

L'ereditarietà

Obiettivi

- Imparare a creare nuove classi a partire da classi già esistenti grazie all'ereditarietà
- Comprendere l'importanza dell'ereditarietà nello sviluppo di software riutilizzabile
- Comprendere i concetti di classe base e classe derivata
- Comprendere l'ereditarietà multipla che permette di derivare una classe da più classi base

9.1 Introduzione

In questo capitolo e in quello seguente parleremo di due funzionalità importanti della programmazione orientata agli oggetti: l'ereditarietà e il polimorfismo. L'ereditarietà è fondamentale nella creazione di software riutilizzabile, perché le nuove classi sono progettate sulla base di classi già esistenti: le nuove classi assimilano gli attributi e i comportamenti delle classi esistenti, sovrascrivendo e migliorando le caratteristiche di quest'ultime secondo le nuove necessità. Come potete intuire, ciò comporta un notevole risparmio di tempo in fase di progettazione e di implementazione e, inoltre, il riutilizzo di software di alta qualità già testato contribuisce a ridurre il numero di problemi che si presentano quando si mettono a punto le nuove classi. Il polimorfismo, di cui parleremo nel prossimo capitolo, consente poi di scrivere i programmi in modo generale, per trattare un'ampia gamma di classi correlate, già esistenti o persino non ancora specificate. L'ereditarietà e il polimorfismo costituiscono due tecniche fondamentali per gestire la complessità del software.

Quando create una nuova classe, anziché scrivere per esteso nuovi dati e funzioni membro, potete decidere che essa *erediti* dati e funzioni membro di una *classe base* già definita. La nuova classe prende il nome di *classe derivata*. Una classe derivata può diventare essa stessa classe base per una classe futura. Nell'*ereditarietà singola* si ha una nuova classe che deriva da una sola classe base mentre nell'*ereditarietà multipla* la nuova classe deriva da più classi base, le quali possono anche non essere in alcun modo correlate. L'ereditarietà singola è semplice e diretta: ne daremo molti esempi, per cui pensiamo che riuscirete a padroneggiarla abbastanza presto. Al contrario l'ereditarietà multipla è piuttosto complessa ed è molto facile commettere errori nell'utilizzarla: mostreremo un solo esempio di ereditarietà multipla, incoraggiandovi a studiare l'argomento più a fondo prima di farne uso nei vostri programmi.

Una classe derivata può prevedere propri dati e funzioni membro oltre a quelli ereditati dalla classe base, per cui le dimensioni delle due classi possono non coincidere. La classe derivata è più specifica della classe base da cui deriva e rappresenta un insieme di oggetti più piccolo (con meno elementi). Nell'ereditarietà singola, la classe derivata non è altro che una versione della classe base: le potenzialità di questo tipo di ereditarietà nascono dal fatto che nella nuova classe si possono definire aggiunte, sostituzioni o miglioramenti alle caratteristiche presenti nella classe base.

Il C++ prevede tre tipi di ereditarietà: `public`, `protected` e `private`. In questo capitolo ci soffermeremo sull'ereditarietà di tipo `public`, mentre degli altri due tipi daremo solo una spiegazione sommaria. Nel Capitolo 4 del volume Tecniche Avanzate illustreremo come l'ereditarietà di tipo `private` può diventare una forma alternativa di composizione. Il terzo tipo di ereditarietà, `protected`, è una novità abbastanza recente del C++ ed è utilizzato raramente. Nell'ereditarietà di tipo `public` gli oggetti di una classe derivata possono essere visti sostanzialmente anche come oggetti della classe base. Tuttavia non vale il contrario, perché gli oggetti della classe base non sono anche oggetti della classe derivata. L'espressione anglosassone "derived-class-object-is-a-base-class-object" (un oggetto della classe derivata è un oggetto della classe base) esprime proprio questa relazione, che ci permette di effettuare manipolazioni interessanti. Per esempio, possiamo utilizzare un array di elementi della classe base per contenere un'ampia gamma di oggetti diversi. In questo modo possiamo effettuare manipolazioni su oggetti diversi in un modo completamente generale. Come vedremo nel prossimo capitolo, questa funzionalità prende il nome di polimorfismo ed è un'altra delle caratteristiche fondamentali della programmazione orientata agli oggetti.

In questo capitolo studieremo una nuova forma di controllo dell'accesso ai membri di una classe: l'accesso `protected`. Se una classe base possiede dei membri `protected`, possono accedervi solamente le classi derivate e `friend`.

L'esperienza insegna che porzioni significative del codice trattano casi particolari strettamente correlati. Nella progettazione di un sistema diventa difficile avere un'immagine generica del problema, perché il programmatore è concentrato nella risoluzione dei casi particolari. La programmazione orientata agli oggetti cerca in vari modi di "far scorgere la foresta tra un albero e un altro", in un processo che prende il nome di *astrazione*.

Se un programma si occupa di un gran numero di casi particolari, trattando individualmente ognuno di essi sarà generalmente sovraccarico di istruzioni che distinguono un caso dall'altro. Nel Capitolo 10 mostreremo come riscrivere un programma di questo tipo in una forma più semplice.

Ora distinguiamo tra *relazione "is a"* (in inglese "è un") e *relazione "has a"* (in inglese "ha un"). La relazione "is a" è l'ereditarietà; in una relazione di questo tipo un oggetto di una classe derivata può essere trattato anche come oggetto della classe base. La relazione "has a" è la composizione (cfr. Figura 7.4): in questa relazione un oggetto di una classe ha come membri oggetti di altre classi.

Una classe derivata non può accedere ai membri `private` della classe base, in quanto ciò violerebbe l'incapsulamento della classe base. Gli unici membri della classe base che restano accessibili sono i membri `public` e `protected`. I membri della classe base che devono restare inaccessibili alle classi derivate vanno dichiarati dunque come `private` nella classe base. L'unico modo di accedere a quei dati, per una classe derivata, è quello di usare le funzioni dichiarate nelle interfacce `public` e `protected` della classe base.

Un problema che comporta l'ereditarietà sta nel fatto che una classe derivata può ereditare l'implementazione di funzioni membro `public` di cui non ha bisogno, o che non dovrebbe avere. Se l'implementazione di un membro della classe base non è adatta alle caratteristiche della classe derivata, tale membro può essere ridefinito nella classe derivata con una nuova implementazione. In alcuni casi, l'ereditarietà di tipo `public` è semplicemente inappropriata.

Fra le cose più apprezzabili dell'ereditarietà c'è il fatto che si possono definire nuove classi a partire dalle librerie di classi esistenti. Ci sono organizzazioni che creano nuove librerie sfruttando quelle già disponibili. In futuro, la maggior parte del software esistente sarà creato prevalentemente a partire da componenti standardizzati e riutilizzabili come succede oggi per l'hardware. Questo modo di operare assumerà una grande importanza nel futuro, man mano che la necessità di sviluppare software più potente diventerà più stringente.

9.2 Le classi base e le classi derivate

In un certo senso la relazione "is a" è frequente nel mondo reale perché un oggetto di una classe è spesso oggetto anche di altre classi. Per esempio, un rettangolo è sicuramente anche un quadrilatero, così come lo sono i quadrati, i parallelogrammi e i trapezi. Perciò una classe `Rectangle` (rettangolo) può essere derivata dalla classe `Quadrilateral` (quadrilatero); in questo contesto, la classe `Quadrilateral` viene detta *classe base* e la classe `Rectangle` *classe derivata*. Un rettangolo è un tipo specifico di quadrilatero, mentre non è esatto dire che un quadrilatero è un rettangolo: per esempio, potrebbe trattarsi di un parallelogramma. La Figura 9.1 mostra alcuni semplici esempi di ereditarietà.

Altri linguaggi di programmazione orientati agli oggetti, come Smalltalk, usano una diversa terminologia: la classe base viene detta *superclasse* e rappresenta un sovrainsieme di oggetti (l'insieme di livello superiore), mentre la classe derivata viene detta *sottoclasse* e rappresenta un sottoinsieme di oggetti.

Classe base	Classe derivata
`Student` (studente)	`GraduateStudent` (studente laureato, ad es. specializzando) `UndergraduateStudent` (studente di corso di laurea)
`Shape` (forma geometrica)	`Circle` (cerchio) `Triangle` (triangolo) `Rectangle` (rettangolo)
`Loan` (prestito)	`CarLoan` (prestito per l'acquisto di un'automobile) `HomeImprovementLoan` (prestito per una ristrutturazione domestica) `MortgageLoan` (mutuo ipotecario)
`Employee` (dipendente)	`FacultyMember` (docente) `StaffMember` (funzionario)
`Account` (conto)	`CheckingAccount` (conto corrente) `SavingsAccount` (deposito vincolato)

Figura 9.1 Alcuni semplici esempi di ereditarietà.

Dato che normalmente l'ereditarietà crea classi derivate che possiedono maggiori funzionalità rispetto alla classe base, i termini *superclasse* e *sottoclasse* possono generare confusione per cui, nel nostro testo, scegliamo di evitarli. Gli oggetti di una classe derivata possono essere pensati come oggetti delle classi base, *e questo implica che ci sono più* oggetti associati con le classi base e meno oggetti associati con le classi derivate: è in questo senso che si parla delle classi base come di "superclassi" e di quelle derivate come di "*sottoclassi*".

La rappresentazione della relazione di ereditarietà forma degli alberi gerarchici nei *quali le classi* base sono in relazione gerarchica con le proprie classi derivate. Una classe ha, in generale, un'esistenza propria indipendente, ma quando si usa il meccanismo dell'ereditarietà essa può assumere il *ruolo di classe base, che passa i propri* attributi e comportamenti ad altre classi, oppure quello di classe derivata, che eredita queste caratteristiche da altre classi.

Proviamo adesso a creare una semplice gerarchia; una tipica sede universitaria è attraversata ogni giorno da migliaia di persone che ne fanno parte. Esse possono essere suddivise, a seconda del loro ruolo nell'organizzazione, in dipendenti, studenti dei corsi di laurea e studenti di corsi post-laurea. I dipendenti, a loro volta, possono essere divisi in docenti e personale tecnico-amministrativo. I docenti *possono essere dei dirigenti* (come i presidi di facoltà e i direttori di dipartimento) o semplicemente degli insegnanti. Questo discorso è sintetizzato nella gerarchia che illustriamo in Figura 9.2. Notate che alcuni amministratori insegnano anche in aula, per cui abbiamo utilizzato l'ereditarietà multipla per la classe `AdministratorTeacher`. Inoltre spesso gli studenti lavorano per la propria università, così come i dipendenti seguono *dei corsi, per cui ha senso creare la classe* `EmployeeStudent` utilizzando l'ereditarietà multipla.

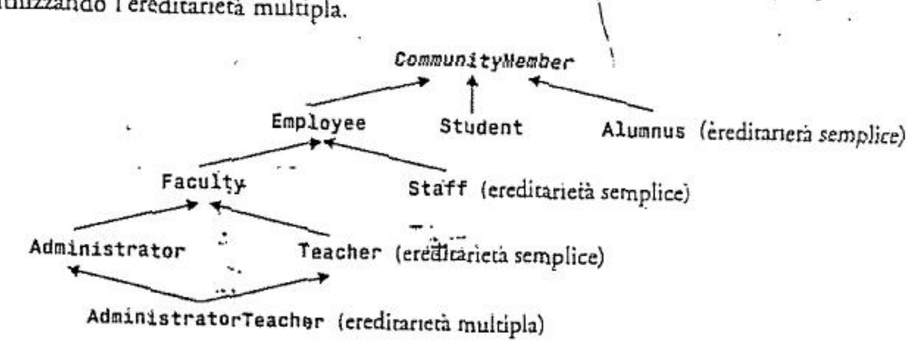


Figura 9.2 Gerarchia dell'ereditarietà tra le persone che frequentano un campus.

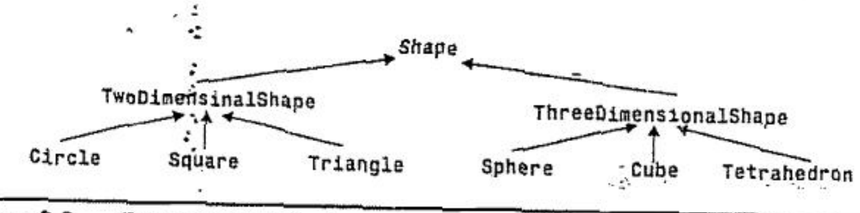


Figura 9.3 Gerarchia parziale di una classe `Shape` (forma geometrica)

Un'altra struttura gerarchica è quella di `Shape` *presentata in Figura 9.3.* Spesso quando si insegna l'ereditarietà nelle classi universitarie, gli studenti si sorprendono del gran numero di gerarchie che si possono trarre dal mondo reale. Il fatto è che spesso questi studenti

non sono abituati a categorizzare il mondo reale in questa maniera, per cui devono fare prima un lavoro su se stessi per imparare a pensare in questi termini.

Adesso passiamo a vedere, in alcuni esempi, la sintassi dei meccanismi di ereditarietà. Per specificare che la classe `CommissionWorker` deriva dalla classe `Employee` possiamo farlo nel modo seguente:

```
class CommissionWorker : public Employee {
   ...
};
```

L'ereditarietà specificata da questa sintassi è di tipo `public`, quello più utilizzato comunemente; in seguito, parleremo anche dell'ereditarietà di tipo `private` e di tipo `protected`. Nell'ereditarietà di tipo `public`, la classe derivata eredita i membri `public` e `protected` della classe base, rispettivamente come `public` e `protected`. Ricordate che i membri `private` della classe base non sono accessibili dalla classe derivata e che le funzioni `friend` non sono ereditate.

Come abbiamo già detto, è possibile trattare allo stesso modo un oggetto della classe derivata (con ereditarietà di tipo `public`) e uno della classe base e questa caratteristica è espressa negli attributi e nei comportamenti della classe base. Daremo diversi esempi in cui sfruttiamo questa proprietà, che permette di semplificare i programmi in modo impensabile in altri linguaggi non orientati agli oggetti.

9.3 I membri protected

Abbiamo visto che i membri `public` di una classe base sono accessibili da tutte le funzioni di un programma mentre i membri `private` sono accessibili soltanto dalle funzioni membro e `friend` della classe base.

La modalità di accesso `protected` si trova a un punto intermedio nel livello di protezione; i membri `protected` di una classe base sono accessibili soltanto da funzioni membro e `friend` di tale classe e di tutte le classi derivate. I membri delle classi derivate possono accedere ai membri `public` e `protected` della classe base semplicemente tramite il loro nome. Osservate che il concetto di dato `protected` va contro il principio di incapsulamento, perché una modifica di un membro `protected` di una classe base potenzialmente potrebbe richiedere la successiva modifica di tutte le classi derivate.

Ingegneria del software 9.1

Come regola generale, dichiarate `private` *i dati membro di una classe e utilizzate la modalità* `protected` *se proprio non potete evitarlo. Questo principio va tenuto presente soprattutto se nella messa a punto di un sistema si deve concentrare l'attenzione sull'efficienza.*

9.4 Il cast dei puntatori a una classe base in puntatori a una classe derivata

Un oggetto di una classe derivata con ereditarietà di tipo `public` può essere trattato come un oggetto della sua classe base e ciò consente delle manipolazioni interessanti. Per esempio, anche se oggetti di classi diverse che discendono da una classe base comune possono *essere notevolmente diversi tra di loro, possiamo ancora creare un array che li contenga:*

infatti, è possibile, a questo scopo, utilizzare un array avente come tipo gli oggetti della classe base. Vi ricordiamo ancora una volta che non vale il contrario: un oggetto di una classe base non è automaticamente anche oggetto di una classe derivata.

Errore tipico 9.1

Se trattate un oggetto di una classe base come se fosse oggetto di una sua classe derivata potreste commettere degli errori.

Il programmatore, in ogni caso, può utilizzare un cast esplicito per convertire un puntatore a una classe base in puntatore a una classe derivata, tuttavia bisogna porre molta attenzione. Infatti, quando si andrà a dereferenziare tale puntatore, si deve essere certi che il suo tipo corrisponda al tipo di oggetto a cui punta. Le tecniche che utilizziamo in questa sezione sono consentite sulla maggior parte dei compilatori attuali. Nello standard ANSI/ISO C++ sono introdotti alcuni accorgimenti per la gestione dinamica dei tipi che preserva da alcuni errori nelle operazioni di dereferenziazione. Nel Capitolo 10 del volume Tecniche Avanzate parleremo proprio di questi argomenti.

Errore tipico 9.2

Se effettuate una conversione di un puntatore a una classe base, rendendolo puntatore a una classe derivata, non potete poi usarlo per riferire membri della classe derivata che non esistono nell'oggetto a cui esso punta: se lo fate si verificheranno degli errori logici durante l'esecuzione.

Il nostro primo esempio è in Figura 9.4. Point.h e point.cpp mostrano la definizione della classe `Point` e delle sue funzioni membro. Circle.h e circle.cpp introducono la definizione della classe `Circle` e delle sue funzioni membro. Fig09_04.cpp presenta un programma di esempio, in cui assegniamo dei puntatori alla classe derivata a puntatori alla classe base ed effettuiamo il cast di puntatori alla classe base in puntatori alla classe derivata.

```
1  // Fig. 9.4: point.h
2  // Definizione della classe Point
3  #ifndef POINT_H
4  #define POINT_H
5
6  class Point {
7     friend ostream &operator<<( ostream &, const Point & );
8  public:
9     Point( int = 0, int = 0 );       // costruttore di default
10    void setPoint( int, int );       // imposta le coordinate
11    int getX() const { return x; }   // funzione get per la x
12    int getY() const { return y; }   // funzione get per la y
13 protected:          // accessibile dalle classi derivate
14    int x, y;        // coordinate x,y del Point
15 };
16
17 #endif
```

Figura 9.4 Cast di puntatori alla classe base in puntatori alla classe derivata (continua)


```
18 // Fig. 9.4: point.cpp
19 // Funzioni membro della classe Point
20 #include <iostream.h>
21 #include "point.h"
22
23 // Costruttore della classe Point
24 Point::Point( int a, int b ) { setPoint( a, b ); }
25
26 // Imposta le coordinate x e y di Point
27 void Point::setPoint( int a, int b )
28 {
29    x = a;
30    y = b;
31 }
32 // Visualizza Point (con l'overloading dell'operatore di
33 // inserimento nello stream)
34 ostream &operator<<( ostream &output, const Point &p )
35 {
36    output << '[' << p.x << ", " << p.y << ']';
37
38    return output;   // consente di scrivere chiamate a cascata
39 }
40 // Fig. 9.4: circle.h
41 // Definizione della classe Circle
42 #ifndef CIRCLE_H
43 #define CIRCLE_H
44
45 #include <iostream.h>
46 #include <iomanip.h>
47 #include "point.h"
48
49 class Circle : public Point {  // Circle deriva da Point
50    friend ostream &operator<<( ostream &, const Circle & );
51 public:
52    // costruttore di default
53    Circle( double r = 0.0, int x = 0, int y = 0 );
54
55    void setRadius( double );   // imposta il raggio
56    double getRadius() const;   // restituisce il raggio
57    double area() const;        // calcola l'area
58 protected:
59    double radius;
60 };
61
62 #endif
```

Figura 9.4 Cast di puntatori alla classe base in puntatori alla classe derivata (continua)

```
63  // Fig. 9.4: circle.cpp
64  // Definizioni delle funzioni membro della classe Circle
65  #include "circle.h"
66
67  // Il costruttore di Circle chiama il costruttore di Point
68  // con un inizializzatore di membro e poi inizializza il raggio.
69  Circle::Circle( double r, int a, int b )
70     : Point( a, b )      // chiama il costruttore della classe base
71  { setRadius( r ); }
72
73  // Imposta il raggio di Circle
74  void Circle::setRadius( double r )
75     { radius = ( r >= 0 ? r : 0 ); }
76
77  // Restituisce il raggio di Circle
78  double Circle::getRadius() const { return radius; }
79
80  // Calcola l'area
81  double Circle::area() const
82     { return 3.14159 * radius * radius; }
83
84  // Visualizza Circle nella forma:
85  // Center = [x, y]; Radius = #.##
86  ostream &operator<<( ostream &output, const Circle &c )
87  {
88     output << "Center = " << static_cast< Point >( c )
89            << "; Radius ="
90            << setiosflags( ios::fixed | ios::showpoint )
91            << setprecision( 2 ) << c.radius;
92
93     return output;    // consente di scrivere chiamate a cascata
94  }
95  // Fig. 9.4: fig09_04.cpp  Cast di puntatori alla classe
96  // base in puntatori alla classe derivata
97  #include <iostream.h>
98  #include <iomanip.h>
99  #include "point.h"
100 #include "circle.h"
101
102 int main()
103 {
104    Point *pointPtr = 0, p( 30, 50 );
105    Circle *circlePtr = 0, c( 2.7, 120, 89 );
106
107    cout << "Point p: " << p << "\nCircle c: " << c << '\n';
108 // Tratta un Circle come Point (vede soltanto
109 // la porzione di classe base)
```

Figura 9.4 Cast di puntatori alla classe base in puntatori alla classe derivata (continua)


```
110    pointPtr = &c;   // assegna l'indirizzo di Circle a pointPtr
111    cout << "\nCircle c (via *pointPtr): "
112         << *pointPtr << '\n';
113
114 // Tratta un Circle come Circle (con qualche cast)
115    pointPtr = &c;   // assegna l'indirizzo di Circle a pointPtr
116 // cast di un puntatore alla classe base in puntatore
117 // alla classe derivata
118    circlePtr = static_cast< Circle * >( pointPtr );
119    cout << "\nCircle c (via *circlePtr):\n" << *circlePtr
120         << "\nArea of c (via circlePtr): "
121         << circlePtr->area() << '\n';
122
123 // PERICOLO: Tratta un Point come Circle
124    pointPtr = &p;   // assegna l'indirizzo di Point a pointPtr
125 // cast di un puntatore alla classe base in puntatore
126 //  alla classe derivata
127    circlePtr = static_cast< Circle * >( pointPtr );
128    cout << "\nPoint p (via *circlePtr):\n" << *circlePtr
129         << "\nArea of object circlePtr points to: "
130         << circlePtr->area() << endl;
131    return 0;
132 }
```

```
Point p: [30, 50]
Circle c: Center = [120, 89]; Radius = 2.70

Circle c (via *pointPtr): [120, 89]

Circle c (via *circlePtr):
Center = [120, 89]; Radius = 2.70
Area of c (via circlePtr): 22.90

Point p (via *circlePtr):
Center = [30, 50]; Radius = 0.00
Area of object circlePtr points to: 0.00
```

Figura 9.4 Cast di puntatori alla classe base in puntatori alla classe derivata.

Prendiamo in esame per prima la definizione della classe `Point` (Figura 9.4, point.h). L'interfaccia `public` di `Point` include le funzioni membro `setPoint`, `getX` e `getY` mentre i dati membro `x` e `y` sono dichiarati come `protected`. Ciò significa che i client di `Point` non potranno accedervi direttamente, mentre le classi derivate di `Point` potranno accedere a tali dati ricevuti grazie all'ereditarietà. Se i dati fossero `private`, per accedervi si dovrebbero utilizzare le funzioni membro `public` di `Point`, e ciò varrebbe anche per le classi derivate. Osservate che l'overloading di `Point` dell'operatore di inserimento nello stream (Figura 9.4, point.cpp) può riferire le variabili `x` e `y` direttamente perché tale funzione è `friend` della classe `Point`. Osservate, inoltre, che `x` e `y` devono essere riferiti tramite un oggetto, come in `p.x` e `p.y`: questo perché la funzione operatore non è membro di `Point` e, dunque, dobbiamo utilizzare un handle esplicito per informare il compilatore

su quale oggetto vogliamo manipolare. Notate che questa classe contiene funzioni membro `inline` come `getX` e `getY`, così che `operator<<` non debba essere necessariamente dichiarata come `friend` per guadagnare in efficienza. Tuttavia non è sempre possibile fornire tutte le funzioni membro necessarie nell'interfaccia `public` di ogni classe, per cui spesso è appropriato usare funzioni `friend`.

La classe `Circle` (Figura 9.4, circle.h) deriva dalla classe `Point` con ereditarietà di tipo `public`. Ciò viene specificato nella prima linea della definizione di classe

```
Class Circle : public Point {  // Circle deriva da Point
```

Il carattere due punti (`:`) nell'intestazione segnala appunto l'ereditarietà e la parola chiave `public` indica il tipo di ereditarietà in questione. Introdurremo gli altri due tipi di ereditarietà nella Sezione 9.7. Tutti i membri `public` e `protected` della classe `Point` sono ereditati rispettivamente come `public` e `protected` dalla classe `Circle`. Ciò significa che l'interfaccia public di `Circle` possiede i membri `public` di `Point`, oltre ai membri `public` che sono propri di `Circle`, cioè `area`, `setRadius` e `getRadius`.

Il costruttore di `Circle` (Figura 9.4, circle.cpp) deve chiamare il costruttore di `Point` per inizializzare la porzione di `Point` che è contenuta in un oggetto `Circle`; esso effettua questa operazione tramite un inizializzatore di membro (cfr. Capitolo 7), nel modo seguente:

```
Circle::Circle( double r, int a, int b)
   : Point( a, b )        // chiama il costruttore della classe base
```

La seconda linea invoca il costruttore di `Point` passandogli i valori `a` e `b` per inizializzare i membri `x` e `y` della classe base. Se il costruttore di `Circle` non chiamasse esplicitamente il costruttore di `Point`, verrebbe effettuata una chiamata automatica al costruttore di default di `Point`, che imposterebbe i valori di default in `x` e `y` alla coppia (0, 0). Nel nostro caso, se la classe `Point` non prevedesse un costruttore di default, il compilatore segnalerebbe un errore di sintassi. Notiamo che si può inviare in output la porzione `Point` di un oggetto `Circle` tramite la funzione `operator<<`, dichiarata nella classe `Circle`, effettuando un cast del riferimento `c` di tipo `Circle` nel tipo `Point`. Si ha una chiamata alla funzione `operator<<` di `Point`, che invia in output le coordinate `x` e `y` con la formattazione di un oggetto `Point`.

Il programma (Figura 9.4, Fig09_04.cpp) definisce `pointPtr` come puntatore a un oggetto `Point`, e crea `p` come istanza della classe `Point`; quindi definisce `circlePtr` come puntatore a un oggetto `Circle`, e crea `c` come istanza di `Circle`. Gli oggetti `p` e `c` sono inviati in output tramite gli operatori di inserimento nello stream ridefiniti nelle rispettive classi, al solo scopo di segnalare che sono stati inizializzati correttamente. In seguito si assegna un puntatore alla classe derivata (l'indirizzo dell'oggetto `c`) al puntatore alla classe base `pointPtr`, inviando in output l'oggetto tramite la funzione `operator<<` di `Point` e il puntatore dereferenziato `*pointPtr`. Notiamo che viene visualizzata soltanto la porzione `Point` dell'oggetto `c` (che è di tipo `Circle`). Nell'ereditarietà di tipo `public` è sempre possibile assegnare un puntatore a una classe derivata a un puntatore a una classe base, perché un oggetto di una classe derivata è anche un oggetto della classe base. Il puntatore alla classe base "vede" soltanto la parte relativa alla classe base nell'oggetto della classe derivata. Il compilatore provvede a convertire implicitamente il puntatore alla classe derivata in puntatore alla classe base.

Nel seguito del programma troviamo l'assegnamento di un puntatore alla classe derivata (l'indirizzo dell'oggetto `c`) al puntatore alla classe base `pointPtr`, e il cast di `pointPtr` nuovamente nel tipo `Circle *`.

Il risultato dell'operazione di cast viene poi assegnato a `circlePtr`. L'oggetto `c` viene inviato in output tramite l'operatore per l'inserimento nello stream, ridefinito nella classe `Circle`, e il puntatore dereferenziato `*circlePtr`. L'area dell'oggetto `c` viene ottenuta inviando un messaggio a `circlePtr`. Il suo valore risulta valido perché i vari puntatori per cui `c` è passato hanno sempre puntato a un oggetto `Circle`.

Un puntatore a una classe base, al contrario, non può essere assegnato direttamente ad un puntatore a una classe derivata, perché in tale assegnamento è insito un pericolo: si suppone sempre che i puntatori a una classe derivata puntino a oggetti della classe derivata e il compilatore non è in grado di effettuare una conversione implicita. Se utilizziamo un cast esplicito, invece, informiamo il compilatore di conoscere il pericolo intrinseco in questo tipo di conversione, per cui ci assumiamo anche la responsabilità di utilizzare il puntatore nel modo corretto: per questo motivo il compilatore permette questa conversione, sebbene rischiosa.

Nel seguito del programma troviamo l'assegnamento di un puntatore alla classe base (l'indirizzo dell'oggetto `p`) al puntatore alla classe base `pointPtr`, e il cast di `pointPtr` nuovamente nel tipo `Circle *`. Il risultato di tale operazione di cast viene poi assegnato a `circlePtr`. L'oggetto `p` viene inviato in output tramite la funzione `operator<<` di `Circle` attraverso il puntatore dereferenziato `*circlePtr`. Il valore del raggio che rileviamo nell'output è nullo (il raggio in realtà non esiste perché `circlePtr` si riferisce a un oggetto `Point`). Se si tratta un oggetto `Point` come se fosse un oggetto `Circle` si ha un valore di `radius` indefinito (il caso ha voluto che ci fosse il valore zero): i puntatori hanno sempre puntato a un oggetto `Point`, e un oggetto `Point` non ha un membro `radius`. Perciò il programma invia in output qualsiasi valore sia presente in memoria nell'indirizzo che si ritiene riservato al membro `radius`, a partire dall'indirizzo conservato in `circlePtr`. Anche l'area dell'oggetto a cui punta `circlePtr` (l'oggetto `p`) viene inviata in output tramite `circlePtr`. Notiamo come il valore dell'area sia `0.00` perché il calcolo si basa sul valore indefinito di `radius`. Ovviamente l'accesso a dati inesistenti comporta dei rischi e le chiamate a funzioni membro inesistenti possono provocare il blocco del programma.

In questa sezione abbiamo discusso il funzionamento delle conversioni tra puntatori. Abbiamo posto quindi le basi necessarie per il prossimo capitolo, in cui approfondiremo la trattazione della programmazione orientata agli oggetti e del polimorfismo.

9.5 Utilizzo delle funzioni membro

A volte le funzioni membro di una classe derivata devono poter accedere a dati e funzioni membro della classe base.

Ingegneria del software 9.2

Una classe derivata non può accedere direttamente ai membri `private` della propria classe base.

Ciò costituisce un punto cruciale nello sviluppo del software in C++: se una classe derivata potesse accedere ai membri `private` della propria classe base si violerebbe l'incapsulamento della classe base. Mantenere nascosti i membri `private` è poi di grande

aiuto nella verifica e nel debugging dei sistemi. Se una classe derivata potesse accedere ai membri `private` della propria classe base, l'accesso a quei dati sarebbe possibile anche per le classi derivate da tale classe derivata, e via dicendo. Ciò costituirebbe una propagazione dell'accesso a dati che si considerano privati, disperdendo i benefici dell'incapsulamento lungo l'intera gerarchia delle classi.

9.6 L'overriding di membri della classe base in una classe derivata

Una classe derivata può *effettuare l'overriding* (sovrascrittura) di una funzione membro della classe base fornendone una nuova versione con nome e segnatura identici (se le segnature sono differenti si parla di *overloading* piuttosto che di *overriding*). Quando si indica il nome di tale funzione nella classe derivata, viene selezionata automaticamente la versione ridefinita nella classe derivata. È possibile accedere alla versione della classe base dall'interno della classe derivata utilizzando l'operatore di risoluzione dello scope.

Errore tipico 9.3

Se si effettua l'overriding di una funzione membro della classe base in una classe derivata, in molti casi la versione della classe derivata chiama prima la versione della classe base e poi esegue altre operazioni. Se non si utilizza l'operatore di risoluzione dello scope per riferire la funzione membro della classe base si verificherà un problema di ricorsione infinita, perché la funzione membro della classe derivata continua a chiamare se stessa. Alla fine ciò causa un esaurimento della memoria di sistema, e quindi un errore fatale durante l'esecuzione.

Consideriamo una semplice classe come `Employee`. Le informazioni che essa contiene, ad esempio i nomi degli impiegati in `firstName` (nome) e `lastName` (cognome), sono necessarie per tutti gli impiegati, anche quelli delle classi che derivano da `Employee`. Deriviamo ora da `Employee` le classi `HourlyWorker`, `PieceWorker`, `Boss` e `CommissionWorker`. Un impiegato `HourlyWorker` viene retribuito con una paga oraria: tutte le ore di straordinario oltre la 40ma ora settimanale vengono retribuite però con una maggiorazione del 50% rispetto alla paga base. Un impiegato `PieceWorker` viene retribuito con una paga fissa per ogni unità di prodotto: supponiamo per semplicità che egli produca soltanto un tipo di prodotto, così possiamo ridurre i dati membro `private` al solo numero di unità prodotte e alla paga unitaria per prodotto. Un `Boss` ha un salario fisso settimanale. Un impiegato di tipo `CommissionWorker` guadagna invece un piccolo fisso settimanale più una percentuale di commissione per le sue vendite all'ingrosso relative a una data settimana. Per semplicità studieremo soltanto la classe `Employee` e la classe derivata `HourlyWorker`.

La Figura 9.5 illustra il nostro esempio. I file employ.h e employ.cpp mostrano la definizione della classe `Employee` e le definizioni delle sue funzioni membro. I file hourly.h e hourly.cpp mostrano la definizione della classe `HourlyWorker` e la definizione delle funzioni membro di `HourlyWorker`. Fig09_05.cpp mostra un programma di prova per la gerarchia di ereditarietà `Employee / HourlyWorker`: esso crea semplicemente l'istanza di un oggetto `HourlyWorker`, la inizializza e chiama la funzione membro `print` di `HourlyWorker` per inviare in output i dati dell'oggetto.


```
1  // Fig. 9.5: employ.h
2  // Definizione della classe Employee
3  #ifndef EMPLOY_H
4  #define EMPLOY_H
5
6  class Employee {
7  public:
8     Employee( const char *, const char * );  // costruttore
9     void print() const;  // visualizza nome e cognome
10    ~Employee();         // distruttore
11 private:
12    char *firstName;     // stringa allocata dinamicamente
13    char *lastName;      // stringa allocata dinamicamente
14 };
15
16 #endif
17 // Fig. 9.5: employ.cpp
18 // Definizioni delle funzioni membro della classe Employee
19 #include <string.h>
20 #include <iostream.h>
21 #include <assert.h>
22 #include "employ.h"
23
24 // Il costruttore alloca spazio dinamicamente per il
25 // nome e il cognome e utilizza strcpy per copiarli
26 // nell'oggetto.
27 Employee::Employee( const char *first, const char *last )
28 {
29    firstName = new char[ strlen( first ) + 1 ];
30    assert( firstName != 0 ); //term.se lo spazio non è stato all.
31    strcpy( firstName, first );
32
33    lastName = new char[ strlen( last ) + 1 ];
34    assert( lastName != 0 ); //term. se lo spazio non è stato all.
35    strcpy( lastName, last );
36 }
37
38 // Visualizza il nome dell'impiegato
39 void Employee::print() const
40    { cout << firstName << ' ' << lastName; }
41
42 // Il distruttore dealloca la memoria allocata dinamicamente
43 Employee::~Employee()
44 {
45    delete [] firstName; //restituisce la mem. occ.ta da firstName
46    delete [] lastName;  //restituisce la mem. occ.ta da lastName
47 }
```

Figura 9.5 Overriding di una funzione membro di una classe base in una classe derivata (continua)

```
48  // Fig. 9.5: hourly.h
49  // Definizione della classe HourlyWorker
50  #ifndef HOURLY_H
51  #define HOURLY_H
52
53  #include "employ.h"
54
55  class HourlyWorker : public Employee {
56  public:
57     HourlyWorker( const char*, const char*, double, double );
58     double getPay() const;  // calcola e restituisce il salario
59     void print() const;     // overriding della funzione print
60  private:
61     double wage;            // paga oraria
62     double hours;           // ore di lavoro della settimana
63  };
64
65  #endif

66  // Fig. 9.5: hourly.cpp
67  // Definizioni delle funzioni membro della classe HourlyWorker
68  #include <iostream.h>
69  #include <iomanip.h>
70  #include "hourly.h"
71
72  // Costruttore della classe HourlyWorker
73  HourlyWorker::HourlyWorker( const char *first,
74                              const char *last,
75                              double initHours, double initWage )
76     : Employee( first, last ) //chiama il costr. della classe base
77  {
78     hours = initHours;  // i valori andrebbero
79     wage = initWage;    // convalidati
80  }
81
82  // Restituisce la paga di HourlyWorker
83  double HourlyWorker::getPay() const
84  {return hours<40 ? wage*hours: 40*wage + (hours-40)* wage *1.5; }
85  // Visualizza il nome e la paga di HourlyWorker
86  void HourlyWorker::print() const
87  {
88     cout << "HourlyWorker::print() is executing\n\n";
89     Employee::print();//chiama la funzione print della classe base
90
91     cout << " is an hourly worker with pay of $"
92          << setiosflags( ios::fixed | ios::showpoint )
93          << setprecision( 2 ) << getPay() << endl;
94  }
```

Figura 9.5 Overriding di una funzione di una classe base in una classe derivata (continua)


```
95  // Fig. 9.5: fig.09_05.cpp
96  // Overriding di una funzione membro della classe base in
97  // una classe derivata.
98  #include <iostream.h>
99  #include "hourly.h"
100
101 int main()
102 {
103    HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
104    h.print();
105    return 0;
106 }
```

```
HourlyWorker::print() is executing

Bob Smith is an hourly worker with pay of $400.00
```

Figura 9.5 Overriding di una funzione membro di una classe base in una classe derivata.

La definizione della classe `Employee` (Figura 9.5, employ.h) consta di due dati membro di tipo `char *` di nome `firstName` e `lastName`, e tre funzioni membro: un costruttore, un distruttore e la funzione `print`. Il costruttore (Figura 9.5, employ.cpp) riceve due stringhe e alloca dinamicamente gli array di caratteri per memorizzare le stringhe. Utilizziamo la macro `assert` per determinare durante l'esecuzione se è stata effettivamente allocata memoria per `firstName` e `lastName`. Se così non fosse, il programma terminerebbe con un messaggio di errore che indica la condizione che non si è verificata, il numero di linea in cui appare e il file in cui si trova. La convenzione di far restituire il valore `0` dall'operatore `new` in caso non sia possibile allocare la memoria è tuttora supportato da tutti i compilatori. Tuttavia, secondo lo standard del C++, l'operatore `new` dovrebbe lanciare un'eccezione se la memoria non fosse sufficiente: discuteremo di ciò nel Capitolo 2 del volume Tecniche Avanzate. I dati di `Employee` sono `private`, per cui l'unico modo per accedere a essi è la funzione `print`, che invia semplicemente in output il nome e il cognome dell'impiegato. Il distruttore restituisce al sistema la memoria allocata dinamicamente (evitando così una perdita di memoria).

La classe `HourlyWorker` (Figura 9.5, hourly.h) deriva dalla classe `Employee`, con ereditarietà di tipo `public`. Come in precedenza, ciò viene specificato nella prima linea della definizione di classe con il segno di due punti (`:`), come segue:

```
class HourlyWorker : public Employee
```

L'interfaccia `public` di `HourlyWorker` contiene la funzione `print` di `Employee` e le funzioni `getPay` e `print`, membri di `HourlyWorker`. Notate come la classe `HourlyWorker` *definisca la propria funzione* `print` *con lo stesso prototipo di* `Employee::print()`: *questo* è un esempio di overriding. La classe `HourlyWorker` può accedere a entrambe le funzioni `print`. Essa contiene anche i dati membro `wage` e `hours` (che sono dati `private`), utili nel calcolo del salario settimanale degli impiegati.

Il costruttore di `HourlyWorker` (Figura 9.5, hourly.cpp) utilizza la sintassi dell'inizializzatore di membro per passare le stringhe `first` e `last` al costruttore di `Employee`, in modo tale che i membri della classe base possano essere inizializzati, e successivamente inizializza i membri `hours` e `wage`. La funzione membro `getPay` calcola il salario di un impiegato `HourlyWorker`.

La funzione `print` di `HourlyWorker` è un esempio di overriding della funzione `print` di `Employee`. Accade spesso di dover effettuare in una classe derivata l'overriding sulle funzioni della classe base per fornire funzionalità ulteriori e le nuove funzioni spesso si servono della funzione della classe base per effettuare parte del lavoro. Nel nostro esempio la funzione della classe derivata `print` chiama la funzione della classe base per visualizzare il nome dell'impiegato: la funzione `print` della classe base è l'unica a poter accedere a tali dati perché essi sono `private`. La *funzione della classe derivata* `print` *invia poi in output* anche la paga dell'impiegato. La versione di `print` della classe base viene chiamata in questo modo:

```
Employee::print();
```

I nomi e le dichiarazioni (le segnature) delle funzioni, nella classe base e in quella derivata, sono uguali: perciò per chiamare la funzione della classe base occorre precedere il suo nome con il nome della classe e con l'operatore di risoluzione dello scope. Diversamente sarebbe chiamata la funzione della classe derivata, causando una ricorsione infinita (in questo caso la funzione `print` di `HourlyWorker` chiamerebbe se stessa).

9.7 Ereditarietà di tipo public, protected e private

Quando si deriva una nuova classe da una classe base, quest'ultima può essere ereditata come `public`, `protected` o `private`. L'uso dell'ereditarietà di tipo `protected` o `private` è abbastanza raro e occorre prestare molta attenzione nell'utilizzarle: in questo libro utilizziamo normalmente l'ereditarietà di tipo `public` che è sufficiente per una vasta gamma di scopi. La Figura 9.6 riassume le varie possibilità di accesso ai membri della classe base dalla classe derivata, per ciascun tipo di ereditarietà. Nella prima colonna troviamo gli specificatori di accesso ai membri della classe base. Se si deriva una classe da una classe base `public`, i membri `public` della classe base diventano membri `public` della classe derivata, così come i membri `protected` della classe base diventano membri `protected` della classe derivata. I membri `private` della classe base non sono mai accessibili direttamente da una classe derivata: se si vuole accedere a essi occorre utilizzare i membri `public` e `protected` della classe base.

Quando si deriva una nuova classe da una classe base `protected`, i membri `public` e `protected` della classe base diventano membri `protected` della classe derivata. Quando la si deriva da una classe base `private`, i membri `public` e `protected` della classe base diventano membri `private` della classe derivata (per es. le funzioni diventano funzioni di utilità). Le ereditarietà di tipo `private` e `protected` non realizzano una relazione di tipo "is a".

Specificatore di accesso ai membri della classe base	Tipo di ereditarietà		
	`public`	`protected`	`private`
`public`	`public` nella classe derivata Accessibili direttamente da tutte le funzioni membro non `static`, le funzioni `friend` e le funzioni non membro	`protected` nella classe derivata Accessibili direttamente da tutte le funzioni membro non `static` e le funzioni `friend`	`private` nella classe derivata Accessibili direttamente da tutte le funzioni membro non `static` e le funzioni `friend`
`protected`	`protected` nella classe derivata Accessibili direttamente da tutte le funzioni membro non `static`, le funzioni `friend`	`protected` nella classe derivata Accessibili direttamente da tutte le funzioni membro non `static`, le funzioni `friend`	`private` nella classe derivata Accessibili direttamente da tutte le funzioni membro non `static`, le funzioni `friend`
`private`	Nascoste nella classe derivata Accessibili da funzioni membro non `static` e funzioni friend tramite funzioni membro `public` e `protected` della classe base	Nascoste nella classe derivata Accessibili da funzioni membro non `static` e funzioni `friend` tramite funzioni membro `public` e `protected` della classe base	Nascoste nella classe derivata Accesibili da funzioni non `static` e funzioni `friend` tramite funzioni membro `public` e `protected` della classe base

Figura 9.6 Riepilogo delle modalità di accesso ai membri di una classe base dall'interno di una classe derivata.

9.8 Classi base dirette e indirette

Una classe base può essere una classe base diretta o indiretta di una classe derivata. Una classe base diretta è elencata esplicitamente nella dichiarazione di classe, cioè nell'intestazione della classe derivata dopo il segno di due punti (`:`), mentre una classe base indiretta non è elencata esplicitamente nell'intestazione della classe derivata. In quest'ultimo caso la classe base si trova due o più livelli al di sopra della classe derivata nella gerarchia delle classi.

9.9 Utilizzo dei costruttori e dei distruttori nelle classi derivate

Una classe derivata eredita i membri della propria classe base e, dunque, quando si istanzia un oggetto di una classe derivata occorre chiamare il costruttore della classe base per inizializzarli. È possibile fornire nel costruttore della classe derivata un inizializzatore della classe base (con la sintassi dell'inizializzatore di membro descritta in precedenza) per chiamare esplicitamente il costruttore della classe base; in caso contrario il costruttore della classe derivata chiamerà implicitamente il costruttore di default della classe base.

I costruttori e gli operatori di assegnamento della classe base non sono ereditati dalla classe derivata; resta la possibilità, però, per i costruttori e gli operatori di assegnamento della classe derivata di invocare quelli della classe base.

Il costruttore di una classe derivata chiama sempre il costruttore della propria classe base per inizializzare i membri della classe base presenti nella classe derivata. Se si omette il costruttore per una classe derivata, il costruttore di default di tale classe chiama il costruttore di default della classe base. I distruttori sono chiamati in ordine inverso rispetto ai costruttori, per cui il distruttore di una classe derivata viene chiamato prima del distruttore della propria classe base.

Ingegneria del software 9.3

Supponiamo di creare un oggetto di una classe derivata, con entrambe le classi, base e derivata, contenenti oggetti di altre classi. Quando si crea un tale oggetto, sono eseguiti prima i costruttori degli oggetti membro della classe base, quindi il costruttore della classe base, poi i costruttori degli oggetti membro della classe derivata e infine il costruttore della classe derivata. I distruttori sono chiamati in ordine inverso rispetto ai costruttori corrispondenti.

Ingegneria del software 9.4

L'ordine in cui sono costruiti gli oggetti membro è lo stesso in cui essi sono dichiarati nella definizione di classe. L'ordine in cui sono elencati gli inizializzatori di membro non influenza l'ordine di chiamata dei costruttori. Alcuni compilatori, comunque, avvertono il programmatore che l'ordine degli inizializzatori di membro non rispecchia quello reale.

Ingegneria del software 9.5

I costruttori della classe base sono chiamati nell'ordine in cui l'ereditarietà è specificata nella definizione della classe derivata. L'ordine in cui sono specificati i costruttori della classe base nell'elenco degli inizializzatori di membro della classe derivata non influenza l'ordine di chiamata dei costruttori. Alcuni compilatori, comunque, avvertono il programmatore che l'ordine degli inizializzatori di membro non rispecchia quello reale.

Il programma in Figura 9.7 illustra l'ordine di chiamata di costruttori e distruttori per le classi base e derivata. Il programma sinizia mostrando la semplice classe `Point` che contiene un costruttore, un distruttore e dati membro `x` e `y` di tipo `protected`. Il costruttore e il distruttore visualizzano entrambi l'oggetto `Point` per il quale sono stati invocati.


```
1   // Fig. 9.7: point2.h
2   // Definizione della classe Point
3   #ifndef POINT2_H
4   #define POINT2_H
5
6   class Point {
7   public:
8      Point( int = 0, int = 0 );  // costruttore di default
9      ~Point();     // distruttore
10  protected:       // accessibile dalle classi derivate
11     int x, y;     // coordinate x,y del punto
12  };
13
14  #endif
```

```
15  // Fig. 9.7: point2.cpp
16  // Definizioni delle funzioni membro della classe Point
17  #include <iostream.h>
18  #include "point2.h"
19
20  // Costruttore della classe Point
21  Point::Point( int a, int b )
22  {
23     x = a;
24     y = b;
25
26     cout << "Point  constructor: "
27          << '[' << x << ", " << y << ']' << endl;
28  }
29
30  // Distruttore della classe Point
31  Point::~Point()
32  {
33     cout << "Point  destructor:  "
34          << '[' << x << ", " << y << ']' << endl;
35  }
```

```
36  // Fig. 9.7: circle2.h
37  // Definizione della classe Circle
38  #ifndef CIRCLE2_H
39  #define CIRCLE2_H
40
41  #include "point2.h"
42
43  class Circle : public Point {
44  public:
```

Figura 9.7 Ordine di chiamata di costruttori e distruttori di una classe base e di una classe derivata (continua)

```
45       // costruttore di default
46       Circle( double r = 0.0, int x = 0, int y = 0 );
47
48       ~Circle();
49    private:
50       double radius;
51    };
52
53    #endif
54    // Fig. 9.7: circle2.cpp
55    // Definizioni delle funzioni membro della classe Circle
56    #include "circle2.h"
57
58    // Il costruttore di Circle chiama il costruttore di Point
59    Circle::Circle( double r, int a, int b )
60       : Point( a, b )   // chiama il costruttore della classe base
61    {
62       radius = r; // dovrebbe convalidarlo
63       cout << "Circle constructor: radius is "
64            << radius << " [" << x << ", " << y << ']' << endl;
65    }
66
67    // Distruttore della classe Circle
68    Circle::~Circle()
69    {
70       cout << "Circle destructor:  radius is "
71            << radius << " [" << x << ", " << y << ']' << endl;
72    }
73    // Fig. 9.7: fig09_07.cpp
74    // Mostra quando sono chiamati i costruttori/distruttori
75    // della classe base e della classe derivata.
76    #include <iostream.h>
77    #include "point2.h"
78    #include "circle2.h"
79
80    int main()
81    {
82       // Mostra le chiamate di costruttore e distruttore di Point
83       {
84          Point p( 11, 22 );
85       }
86
87       cout << endl;
88       Circle circle1( 4.5, 72, 29 );
89       cout << endl;
90       Circle circle2( 10, 5, 5 );
```

Figura 9.7 *Ordine di chiamata di costruttori e distruttori di una classe base e di una classe derivata (continua)*


```
91       cout << endl;
92       return 0;
93    }
```

```
Point  constructor: [11, 22]
Point  destructor:  [11, 22]

Point  constructor: [72, 29]
Circle constructor: radius is 4.5 [72, 29]

Point  constructor: [5, 5]
Circle constructor: radius is 10 [5, 5]

Circle destructor:  radius is 10 [5, 5]
Point  destructor:  [5, 5]
Circle destructor:  radius is 4.5 [72, 29]
Point  destructor:  [72, 29]
```

Figura 9.7 Ordine di chiamata di costruttori e distruttori di una classe base e di una *classe derivata.*

Circle2.h e circle2.cpp nella Figura 9.7 mostrano un esempio di classe `Circle` derivata da `Point` con ereditarietà di tipo `public`. La classe `Circle` contiene un costruttore, un *distruttore e il dato membro* `radius` *di tipo* `private`. *Sia il costruttore che il distruttore* visualizzano l'oggetto `Circle` per il quale sono stati invocati. Il costruttore di `Circle` invoca inoltre il costruttore di `Point` con la sintassi dell'inizializzatore di membro e passa i valori `a` e `b`, in modo che possano essere inizializzati i dati membro `x` e `y`.

Fig09_07.cpp mostra il programma di esempio per la gerarchia `Point / Circle`. All'inizio il programma istanzia un oggetto `Point` all'interno di un nuovo blocco in `main`. *L'oggetto entra ed esce immediatamente dallo scope, per cui sono chiamati sia il costruttore* che il distruttore di `Point`. Dopo di che il programma istanzia l'oggetto `circle1` di tipo `Circle`: questa operazione invoca il costruttore di `Point` per effettuare l'output dei valori passati dal costruttore di `Circle` e, quindi, effettua l'output indicato nel costruttore di `Circle`. Subito dopo viene istanziato un altro oggetto `Circle`, `circle2`. Sono quindi chiamati nuovamente costruttore e distruttore di `Point` e `Circle`. Notate che il corpo del costruttore di `Point` è eseguito prima del corpo del costruttore di `Circle`. Si raggiunge quindi la fine di `main`, cosa che causa la chiamata ai distruttori di `circle1` e `circle2`, invocati in ordine inverso rispetto ai costruttori corrispondenti. Questo significa che il distruttore di `Circle` e di `Point` sono chiamati in quest'ordine prima per `circle2` e poi per `circle1`.

9.10 Conversione implicita di un oggetto di una classe derivata in oggetto della classe base

Anche se un oggetto di una classe derivata è ("is a") un oggetto della classe base, le due classi sono tipi di dato differenti. Nell'ereditarietà di tipo `public`, gli oggetti di una classe derivata possono essere trattati come se fossero oggetti della classe base. Questa operazione è possibile perché la classe derivata contiene tutti i membri presenti nella classe base (ricordate che la classe derivata può contenere più membri della classe base). L'assegna-

mento nella direzione opposta non è possibile, perché se si assegnasse un oggetto della classe base a uno di una classe derivata, i membri aggiuntivi della classe derivata resterebbero in uno stato indefinito. Tuttavia, sebbene non sia naturale, questo assegnamento si può effettuare ugualmente ridefinendo l'operatore di assegnamento o del costruttore di conversione (cfr. Capitolo 8).

Errore tipico 9.4

Se assegnate un oggetto di una classe derivata a un oggetto di una classe base ed attraverso quest'ultimo tentate di riferire i membri della classe derivata, commettete un errore di sintassi.

Quello che diremo nel resto di questa sezione a proposito dei puntatori vale anche per i riferimenti.

Nell'ereditarietà di tipo `public`, un puntatore a un oggetto di una classe derivata può essere convertito implicitamente in puntatore a un oggetto della classe base, per la relazione "is a" tra classe derivata e classe base.

Ci sono quattro modi per combinare i puntatori alla classe base e alla classe derivata con gli oggetti della classe base e della classe derivata:

1. Accedere a un oggetto della classe base con un puntatore alla classe base è un'operazione diretta.
2. Accedere a un oggetto della classe derivata con un puntatore alla classe derivata è un'operazione diretta.
3. Accedere a un oggetto della classe derivata con un puntatore alla classe base è un'operazione sicura, perché un oggetto della classe derivata è anche oggetto della classe base. In questo modo si possono riferire soltanto i membri della classe base; se si tenta di accedere a membri propri della classe derivata (tramite il puntatore alla classe base), il compilatore segnalerà un errore di sintassi.
4. Accedere a un oggetto della classe base attraverso un puntatore alla classe derivata provoca un errore di sintassi. Il puntatore alla classe derivata deve prima subire una conversione in puntatore alla classe base.

Errore tipico 9.5

Se effettuate il cast di un puntatore a una classe base in puntatore a una classe derivata potreste commettere degli errori, se tramite tale puntatore tentate di riferire membri che esistono soltanto nella classe derivata.

Anche se può essere comodo trattare oggetti di una classe derivata come oggetti della classe base manipolando tutti gli oggetti unicamente tramite puntatori alla classe base, bisogna fare attenzione. In un libro paga elettronico, per esempio, potremmo voler scandire un vettore di impiegati per calcolare la paga settimanale di ciascuno di essi. Se utilizziamo puntatori alla classe base il programma potrà chiamare soltanto la routine di calcolo della paga per la classe base, ammesso naturalmente che una tale routine esista effettivamente nella classe base. Se vogliamo invocare la giusta routine di calcolo della paga per ciascun oggetto, sia esso di classe base o derivata, unicamente tramite puntatori alla classe base, dobbiamo aspettare fino al Capitolo 10, in cui parleremo delle funzioni virtuali e del polimorfismo.

9.11 Il ruolo dell'ereditarietà nell'ingegneria del software

L'ereditarietà può servire a personalizzare il software già esistente. Ereditando gli attributi e i comportamenti delle classi esistenti e aggiungendone altri nuovi (dove occorre effettuando l'overriding dei comportamenti delle classi base), possiamo far sì che le classi già implementate si adattino ai nostri scopi. In C++ questa operazione non ha bisogno di accedere al codice sorgente delle classi base: l'unica cosa necessaria è che si possa effettuare il linking con il codice oggetto delle classi base. I produttori di software traggono dei vantaggi da questa funzionalità del linguaggio, perché possono scrivere le proprie classi rilasciando agli utenti il solo codice oggetto ed i file di intestazione contenenti le interfaccie delle classi. Gli utenti, dal canto loro, possono rapidamente derivare nuove classi senza dover accedere al codice sorgente di proprietà dei produttori. Ciononostante diversi utenti rinunciano a queste possibilità essendo ancora riluttanti ad includere nei loro programmi codice scritto da altri.

Per uno studente può essere difficile stimare i problemi affrontati da coloro che progettano e implementano software su larga scala. Chi ha una certa esperienza in questo campo sa che la chiave per migliorare il processo di sviluppo di un software è il riutilizzo e la programmazione orientata agli oggetti ed il linguaggio C++ sono indubbiamente due scelte giuste in questo senso.

La disponibilità di librerie di classi esalta le potenzialità dell'ereditarietà; cresce l'interesse verso il C++ e di conseguenza anche quello verso le librerie di classi. La produzione di pacchetti software ha conosciuto incrementi esponenziali con l'introduzione dei personal computer ed ora questo fenomeno sta accadendo con le librerie di classi. Le librerie vendute assieme ai compilatori C++ tendono a essere troppo generiche e, dunque, l'obiettivo è la creazione di librerie di classi per l'intera gamma di applicazioni possibili in modo da riempire questo mercato in continua espansione.

Ingegneria del software 9.6

La creazione di una classe derivata non influenza il codice sorgente né il codice oggetto della classe base: l'ereditarietà preserva l'integrità della classe base.

Una classe base specifica le funzionalità comuni, che saranno ereditate dalle classi derivate. Nel processo di progettazione orientata agli oggetti, il progettista deve ricercare queste funzionalità comuni ed evidenziarle, in modo da decidere quali classi basi conviene progettare. Le classi derivate sono poi delle mere estensioni alle funzionalità principali ereditate dalle classi base.

Ingegneria del software 9.7

In un sistema orientato agli oggetti, le classi sono spesso strettamente correlate. Evidenziate gli attributi e i comportamenti che sembrano comuni e poneteli nelle classi base, quindi progettate le classi derivate utilizzando l'ereditarietà.

Così come il progettista di sistemi procedurali cerca di evitare la proliferazione di funzioni inutili, il progettista di sistemi a oggetti deve cercare di evitare le classi inutili. Infatti ciò potrebbe generare problemi di gestione, facendo venir meno il possibile riutilizzo del software: i potenziali utenti potrebbero non riuscire a venire a capo di una collezione di classi gigantesca per trovare la classe specifica di cui hanno bisogno. Un buon compromes-

so consiste nel creare un numero di classi piuttosto basso, ognuna delle quali fornisce delle funzionalità aggiuntive importanti. Classi di questo tipo possono essere ancora troppo ricche per alcuni utenti, i quali possono ignorare le funzionalità aggiuntive, adattando le classi alle proprie esigenze.

Obiettivo efficienza 9.1

Se l'ereditarietà produce classi più grandi di quanto serve, si avrà uno spreco di memoria e di tempo di elaborazione. Derivate le vostre classi da quelle che sembrano più vicine, in termini di funzionalità, alle vostre esigenze.

Osservate che ci si può disorientare leggendo un insieme di dichiarazioni di classi derivate, perché i membri ereditati non sono visibili ma sono presenti ugualmente nelle classi. Un problema simile riguarda la documentazione delle classi derivate.

Ingegneria del software 9.8

Una classe derivata contiene attributi e comportamenti della classe base, insieme a eventuali attributi e comportamenti aggiuntivi. La classe base può essere compilata indipendentemente dalla classe derivata. Occorre compilare soltanto gli attributi e i comportamenti aggiuntivi della classe derivata perché questi possano essere composti con quelli della classe base e possano formare la classe derivata completa.

Ingegneria del software 9.9

Se si effettuano modifiche su una classe base, non occorre cambiare anche le classi derivate, sempre che le interfacce `public` e `protected` della classe base restino inalterate. Tuttavia le classi derivate potrebbero dover essere ricompilate.

9.12 Composizione ed ereditarietà

Finora abbiamo parlato delle relazioni "*is a*" coinvolte nell'ereditarietà di tipo `public`. Abbiamo anche parlato delle relazioni "*has a*" (nel capitolo precedente ne abbiamo dato alcuni esempi) in cui una classe può contenere altre classi come membri: in questo modo le nuove classi sono create per *composizione* da classi esistenti. Per esempio, date le classi `Employee` (impiegato), `BirthDate` (data di nascita) e `TelephoneNumber` (numero telefonico), è scorretto dire che un `Employee` *is a* `BirthDate` o che un `Employee` *is a* `TelephoneNumber`. È giusto, invece, dire che un `Employee` *has a* `BirthDate` e che un `Employee` *has a* `TelephoneNumber`.

Ingegneria del software 9.10

Se si effettuano modifiche in una classe che è membro di un'altra classe, non occorre cambiare anche quest'ultima, sempre che l'interfaccia `public` della classe membro resti inalterata. Tuttavia la classe di composizione potrebbe dover essere ricompilata.

9.13 Le relazioni "uses a" e "knows a"

L'ereditarietà e la composizione giocano un ruolo fondamentale nel riutilizzo del software, perché permettono di creare nuove classi che hanno funzionalità in comune con classi già esistenti. Esistono altri modi per utilizzare i servizi delle classi. Anche se un oggetto persona non è un'automobile e non contiene un'automobile, un oggetto persona certamente *usa*

(*uses a*) un'automobile. Una funzione usa un oggetto effettuando semplicemente una chiamata a una funzione membro di tale oggetto. Un oggetto può essere *a conoscenza* di un altro oggetto; relazioni di questo tipo costituiscono le cosiddette reti di conoscenze. Un oggetto può contenere un handle di un altro oggetto, in forma di puntatore o di riferimento, per essere a conoscenza della presenza dell'altro oggetto. In questo caso si dice che un oggetto una relazione "*knows a*" (*conosce*) con l'altro oggetto. Questa relazione prende anche il nome di *associazione*.

9.14 Progettazione delle classi Point, Circle e Cylinder

Passiamo adesso all'esercizio fondamentale di questo capitolo e estendiamo la gerarchia punto / cerchio includendo ora anche i cilindri. Progetteremo per prima la classe `Point` (Figura 9.8). Quindi presenteremo un esempio in cui deriviamo da `Point` la classe `Circle` (Figura 9.9). Infine presenteremo un esempio in cui deriviamo da `Circle` la classe `Cylinder` (Figura 9.10). Il listato in Figura 9.8 mostra la classe `Point`. Point2.h contiene la definizione di `Point`. Osservate che i dati membro di `Point` sono `protected`. Perciò, quando deriveremo la classe `Circle` da `Point`, le funzioni membro di `Circle` potranno riferire direttamente le coordinate `x` e `y`, anziché dover utilizzare delle funzioni di accesso migliorando l'efficienza del programma.

In point2.cpp troviamo le definizioni delle funzioni membro di `Point` e in fig09_08.cpp troviamo il programma di esempio che fa uso della classe `Point`. Osservate che `main` deve servirsi delle funzioni di accesso `getX` e `getY` per leggere i valori dei dati membro `x` e `y`, perché sono `protected`: ricordate che i dati membro `protected` sono accessibili soltanto alle funzioni membri e `friend` della classe in questione e di quelle da essa derivate.

```
1  // Fig. 9.8: point2.h
2  // Definizione della classe Point
3  #ifndef POINT2_H
4  #define POINT2_H
5
6  class Point {
7     friend ostream &operator<<( ostream &, const Point & );
8  public:
9     Point( int = 0, int = 0 );       // costruttore di default
10    void setPoint( int, int );       // imposta le coordinate
11    int getX() const { return x; }   // restituisce la coordinata x
12    int getY() const { return y; }   // restituisce la coordinata y
13 protected:         // accessibile alle classi derivate
14    int x, y;       // coordinate del punto
15 };
16
17 #endif
```

Figura 9.8 La classe Point (continua)

```
// Fig. 9.8: point2.cpp
// Funzioni membro della classe Point
#include <iostream.h>
#include "point2.h"

// Costruttore della classe Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Imposta le coordinate x e y
void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

// Visualizza Point
ostream &operator<<( ostream &output, const Point &p )
{
   output << '[' << p.x << ", " << p.y << ']';

   return output;       // consente di scrivere chiamate a cascata
}

// Fig. 9.8: fig09_08.cpp
// Programma di esempio per la classe Point
#include <iostream.h>
#include "point2.h"

int main()
{
   Point p( 72, 115 );   // istanzia l'oggetto-Point p.

   // i dati protected di Point non sono accessibili in main
   cout << "X coordinate is " << p.getX()
        << "\nY coordinate is " << p.getY();

   p.setPoint( 10, 10 );
   cout << "\n\nThe new location of p is " << p << endl;

   return 0;
}
```

```
X coordinate is 72
Y coordinate is 115

The new location of p is [10, 10]
```

Figura 9.8 La classe Point.

Il nostro prossimo esempio è in Figura 9.9 Qui sono riutilizzate le definizioni della classe **Point** e delle sue funzioni membro di Figura 9.8. Nella figura sono visibili la definizione della classe **Circle** e delle sue funzioni membro assieme ad un programma di esempio ed al suo output. Osservate che la classe **Circle** deriva dalla classe **Point** con ereditarietà di tipo **public**: ciò significa che l'interfaccia **public** di **Circle** include le funzioni membro di **Point**, insieme con le funzioni membro di **Circle, setRadius, getRadius** e **area**.

```
// Fig. 9.9: circle2.h
// Definizione della classe Circle
#ifndef CIRCLE2_H
#define CIRCLE2_H

#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<( ostream &, const Circle & );
public:
   // costruttore di default
   Circle( double r = 0.0, int x = 0, int y = 0 );
   void setRadius( double );     // imposta radius
   double getRadius() const;     // restituisce radius
   double area() const;          // calcola l'area
protected:           // accessibile alle classi derivate
   double radius;    // raggio del cerchio
};

#endif

// Fig. 9.9: circle2.cpp
// Definizioni delle funzioni membro della classe Circle
#include <iostream.h>
#include <iomanip.h>
#include "circle2.h"

// Il costruttore di Circle chiama il costruttore di Point
// con un inizializzatore di membro e inizializza radius
Circle::Circle( double r, int a, int b )
   : Point( a, b )      // chiama il costruttore della classe base
{ setRadius( r ); }

// Imposta radius
void Circle::setRadius( double r )
   { radius = ( r >= 0 ? r : 0 ); }

// Restituisce radius
double Circle::getRadius() const { return radius; }
```

Figura 9.9 La classe Circle (continua)

```
40  // Calcola l'area del cerchio
41  double Circle::area() const
42     { return 3.14159 * radius * radius; }
43
44  // Visualizza un cerchio nella forma:
45  // Center = [x, y]; Radius = #.##
46  ostream &operator<<( ostream &output, const Circle &c )
47  {
48     output << "Center = " << static_cast< Point > ( c )
49            << "; Radius = "
50            << setiosflags( ios::fixed | ios::showpoint )
51            << setprecision( 2 ) << c.radius;
52
53     return output;    //  consente di scrivere chiamate a cascata
54  }

55  // Fig. 9.9: fig09_09.cpp
56  // Programma di esempio per la classe Circle
57  #include <iostream.h>
58  #include "point2.h"
59  #include "circle2.h"
60
61  int main()
62  {
63     Circle c( 2.5, 37, 43 );
64
65     cout << "X coordinate is " << c.getX()
66          << "\nY coordinate is " << c.getY()
67          << "\nRadius is " << c.getRadius();
68
69     c.setRadius( 4.25 );
70     c.setPoint( 2, 2 );
71     cout << "\n\nThe new location and radius of c are\n"
72          << c << "\nArea " << c.area() << '\n';
73
74     Point &pRef = c;
75     cout << "\nCircle printed as a Point is: " << pRef << endl;
76
77     return 0;
78  }
```

```
X coordinate is 37
Y coordinate is 43
Radius is 2.5
The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area 56.74

Circle printed as a Point is: [2, 2]
```

Figura 9.9 La classe Circle.

Osservate che la funzione `operator<<` nell'overloading di `Circle` è `friend` della classe `Circle` ed è in grado di inviare in output la porzione `Point` di un oggetto `Circle` effettuando il cast di `c`, riferimento di `Circle`, nel tipo `Point`. Questa operazione chiama la funzione `operator<<` di `Point` e invia in output le coordinate `x` e `y` nella formattazione propria di `Point`.

Il programma di esempio fig09_09.cpp istanzia un oggetto della classe `Circle` e poi utilizza le funzioni *get* per ottenere le informazioni sull'oggetto `Circle`. Osservate di nuovo che `main`, non essendo né funzione membro né `friend` della classe `Circle`, non può accedere direttamente ai membri `protected` di `Circle`. Il programma di esempio utilizza poi le funzioni `setRadius` e `setPoint` per azzerare `radius` e le coordinate del centro del cerchio. Infine il programma inizializza con l'oggetto `Circle` `c` il riferimento `pRef` di tipo `Point &`. In seguito, il programma visualizza `pRef` che, nonostante si riferisca ad un oggetto `Circle`, è visto a tutti gli effetti come oggetto di tipo `Point` e come tale viene visualizzato.

Il nostro ultimo esempio è in Figura 9.10. Qui riutilizziamo le definizioni delle classi `Point` e `Circle` e delle loro funzioni membro, dalle Figura 9.8 e Figura 9.9. Cylindr2.h mostra la definizione della classe `Cylinder`, cylindr2.cpp le definizioni delle funzioni membro di `Cylinder` e fig09_10.cpp è un programma di esempio con il relativo output. Osservate che la classe `Cylinder` deriva dalla classe `Circle` con ereditarietà di tipo `public`. Ciò significa che l'interfaccia `public` di `Cylinder` include le funzioni membro di `Circle` e di `Point`, insieme alle funzioni membro proprie di `Cylinder` `setHeight`, `getHeight`, `area` (in overriding rispetto a quella della classe `Circle`) e `volume`. Osservate che il costruttore di `Cylinder` deve invocare il costruttore della classe base `Circle`, ma non quello della classe base indiretta `Point`.

Il costruttore di ogni classe derivata è responsabile soltanto della chiamata ai costruttori della sua classe base immediata (o delle sue classi base, nel caso di ereditarietà multipla). Osservate inoltre che la funzione `operator<<`, nell'overloading della classe `Cylinder`, è `friend` della classe `Cylinder` e può inviare in output la porzione `Circle` di un oggetto `Cylinder` con un cast di `c`, riferimento `Cylinder`, nel tipo `Circle`. Questa operazione causa una chiamata alla funzione `operator<<` di `Circle` e visualizza le coordinate `x` e `y` insieme con `radius` nella formattazione propria di `Circle`.

Il programma di esempio istanzia un oggetto della classe `Cylinder` e utilizza le funzioni *get* per ottenere le informazioni su di esso. Osservate di nuovo che `main` non è né funzione membro né funzione `friend` della classe `Cylinder` per cui non può accedere direttamente ai dati di `Cylinder`, che sono `protected`. Il programma si serve delle funzioni `setHeight`, `setRadius` e `setPoint` per azzerare `height`, `radius` e le coordinate del cilindro. Infine il programma inizializza con l'oggetto `cyl` di tipo `Cylinder` il riferimento `pRef` di tipo `Point &`. Il programma visualizza quindi `pRef` che, nonostante si riferisca ad un oggetto `Cylinder`, è visto come oggetto `Point` e come tale viene visualizzato. Il programma poi usa di nuovo `cyl` per inizializzare il riferimento `circleRef` di tipo `Circle &` e tenta di visualizzarlo. Nuovamente, nonostante esso si riferisca ad un oggetto `Cylinder` è trattato come un oggetto `Circle` e come tale è visualizzato. Infine viene visualizzata l'`area` dell'oggetto `Circle`.

Questo esempio illustra l'ereditarietà di tipo `public` e il concetto di dato membro `protected`. A questo punto dovreste avere una certa padronanza dei concetti fondamentali dell'argomento. Nel prossimo capitolo mostriamo come programmare con le gerarchie

dell'ereditarietà in un modo più generale, tramite il polimorfismo che, assieme all'astrazione dei dati e l'ereditarietà costituisce uno degli assi portanti della programmazione orientata agli oggetti.

```
1  // Fig. 9.10: cylindr2.h
2  // Definizione della classe Cylinder
3  #ifndef CYLINDR2_H
4  #define CYLINDR2_H
5
6  #include "circle2.h"
7
8  class Cylinder : public Circle {
9     friend ostream &operator<<( ostream &, const Cylinder & );
10
11 public:
12    // costruttore di default
13    Cylinder( double h = 0.0, double r = 0.0,
14              int x = 0, int y = 0 );
15
16    void setHeight( double );    // imposta height
17    double getHeight() const;    // restituisce height
18    double area() const;         // calcola e restituisce l'area
19    double volume() const;       // calcola e restituisce il volume
20
21 protected:
22    double height;               // altezza del Cylinder
23 };
24
25 #endif
```

```
26 // Fig. 9.10: cylindr2.cpp
27 // Definizioni delle funzioni membro e friend
28 // della classe Cylinder.
29 #include <iostream.h>
30 #include <iomanip.h>
31 #include "cylindr2.h"
32
33 // Il costruttore di Cylinder chiama il costruttore di Circle
34 Cylinder::Cylinder( double h, double r, int x, int y )
35    : Circle( r, x, y ) // chiama il costruttore della classe base
36 { setHeight( h ); }
37
38 // Imposta height
39 void Cylinder::setHeight( double h )
40    { height = ( h >= 0 ? h : 0 ); }
41
42 // Restituisce height
43 double Cylinder::getHeight() const { return height; }
```

Figura 9.10 La classe Cylinder (continua)


```
44
45 // Calcola l'area del Cylinder (cioè della superficie)
46 double Cylinder::area() const
47 {
48    return 2 * Circle::area() +
49           2 * 3.14159 * radius * height;
50 }
51
52 // Calcola il volume del Cylinder
53 double Cylinder::volume() const
54    { return Circle::area() * height; }
55
56 // Visualizza le dimensioni del Cylinder
57 ostream &operator<<( ostream &output, const Cylinder &c )
58 {
59    output << static_cast< Circle >( c )
60           << "; Height = " << c.height;
61
62    return output;   // consente di scrivere chiamate a cascata
63 }
```

```
64 // Fig. 9.10: fig09_10.cpp
65 // Programma di esempio per la classe Cylinder
66 #include <iostream.h>
67 #include <iomanip.h>
68 #include "point2.h"
69 #include "circle2.h"
70 #include "cylindr2.h"
71
72 int main()
73 {
74 // crea un oggetto Cylinder
75    Cylinder cyl( 5.7, 2.5, 12, 23 );
76
77 // utilizza le funzioni get per visualizzare il Cylinder
78    cout << "X coordinate is " << cyl.getX()
79         << "\nY coordinate is " << cyl.getY()
80         << "\nRadius is " << cyl.getRadius()
81         << "\nHeight is " << cyl.getHeight() << "\n\n";
82
83 // utilizza le funz. set per modificare gli attributi di Cylinder
84    cyl.setHeight( 10 );
85    cyl.setRadius( 4.25 );
86    cyl.setPoint( 2, 2 );
87    cout << "The new location, radius, and height of cyl are:\n"
88         << cyl << '\n';
89
```

Figura 9.10 La classe Cylinder (continua)

```
90   // visualizza Cylinder come un Point
91      Point &pRef = cyl;   // pRef "pensa" che sia un Point
92      cout << "\nCylinder printed as a Point is: "
93           << pRef << "\n\n";
94
95   // visualizza Cylinder come un Circle
96      Circle &circleRef = cyl;  // circleRef pensa che sia un Circle
97      cout << "Cylinder printed as a Circle is:\n" << circleRef
98           << "\nArea: " << circleRef.area() << endl;
99
100     return 0;
101  }
```

```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location, radius, and height of cyl are:
Center = [2, 2]; Radius = 4.25; Height = 10.00

Cylinder printed as a Point is: [2, 2]

Cylinder printed as a Circle is:
Center = [2, 2]; Radius = 4.25
Area: 56.74
```

Figura 9.10 La classe Cylinder.

9.15 L'ereditarietà multipla

Finora abbiamo parlato dell'ereditarietà singola, in cui ogni classe deriva esattamente da una classe base, ma, abbiamo già detto che è possibile far derivare una classe da più classi base. In questo caso si parla di ereditarietà multipla e la classe derivata eredita i membri di più classi base. Le potenzialità di questa tipo di ereditarietà sono di fondamentale importanza nel riutilizzo del software, ma possono anche causare una serie di problemi di ambiguità.

Buona abitudine 9.1

L'ereditarietà multipla ha delle grosse potenzialità, se utilizzata correttamente. Questo tipo di ereditarietà dovrebbe essere utilizzato quando la relazione "is a" sussiste tra il nuovo tipo di dato e due o più tipi di dato esistenti (ad esempio, il tipo A "is a" un tipo B e il tipo A "is a" un tipo C).

Osservate l'esempio di ereditarietà multipla in Figura 9.11. La classe `Base1` contiene come dato membro intero `protected` di nome `value`. `Base1`, inoltre, contiene un costruttore che imposta `value` e la funzione membro `public getData` che restituisce `value`.

La classe `Base2` è simile alla classe `Base1` tranne per il fatto che contiene il dato membro `protected` di nome `letter` e di tipo `char`. Anche `Base2` ha una funzione membro `getData` di tipo `public` che restituisce il valore di `letter`.

La classe `Derived` deriva da `Base1` e `Base2` con ereditarietà multipla. Essa possiede il dato membro privato `real` di tipo `float`, e la funzione membro `public getReal` che legge il suo valore.

```
1   // Fig. 9.11: base1.h
2   // Definizione della classe Base1
3   #ifndef BASE1_H
4   #define BASE1_H
5
6   class Base1 {
7   public:
8      Base1( int x ) { value = x; }
9      int getData() const { return value; }
10  protected:       // accessibile alle classi derivate
11     int value;    // ereditato dalle classi derivate
12  };
13
14  #endif
```

```
15  // Fig. 9.11: base2.h
16  // Definizione della classe Base2
17  #ifndef BASE2_H
18  #define BASE2_H
19
20  class Base2 {
21  public:
22     Base2( char c ) { letter = c; }
23     char getData() const { return letter; }
24  protected:         // accessibile alle classi derivate
25     char letter;    // ereditato dalle classi derivate
26  };
27
28  #endif
```

```
29  // Fig. 9.11: derived.h
30  // Definizione della classe Derived che eredita
31  // più classi base (Base1 e Base2).
32  #ifndef DERIVED_H
33  #define DERIVED_H
34
35  #include "base1.h"
36  #include "base2.h"
37
```

Figura 9.11 Esempio di ereditarietà multipla (continua)

```
38  // ereditarietà multipla
39  class Derived : public Base1, public Base2 {
40     friend ostream &operator<<( ostream &, const Derived & );
41
42  public:
43     Derived( int, char, double );
44     double getReal() const;
45
46  private:
47     double real;    // dati private della classe derivata
48  };
49
50  #endif
51  // Fig. 9.11: derived.cpp
52  // Definizioni delle funzioni membro della classe Derived
53  #include <iostream.h>
54  #include "derived.h"
55
56  // Il costruttore di Derived chiama i costruttori di Base1 e
57  // Base2. Utilizza gli inizializzatori di membro per chiamare i
58  // costruttori delle classi base
59  Derived::Derived( int i, char c, double f )
60     : Base1( i ), Base2( c ), real ( f ) { }
61
62  // Restituisce il valore di real
63  double Derived::getReal() const { return real; }
64
65  // Visualizza tutti i dati membro di Derived
66  ostream &operator<<( ostream &output, const Derived &d )
67  {
68     output << "     Integer: " << d.value
69            << "\n  Character: " << d.letter
70            << "\nReal number: " << d.real;
71
72     return output;    // consente di scrivere chiamate a cascata
73  }
74  // Fig. 9.11: fig09_11.cpp
75  // Programma di esempio dell'ereditarietà multipla
76  #include <iostream.h>
77  #include "base1.h"
78  #include "base2.h"
79  #include "derived.h"
80
81  int main()
82  {
83     Base1 b1( 10 ), *base1Ptr = 0;  // crea un oggetto Base1
84     Base2 b2( 'Z' ), *base2Ptr = 0; // crea un oggetto Base2
```

Figura 9.11 Esempio di ereditarietà multipla (continua)


```
85     Derived d( 7, 'A', 3.5 );     // crea un oggetto Derived
86
87     // visualizza i dati membro degli oggetti di classe base
88     cout << "Object b1 contains integer " << b1.getData()
89          << "\nObject b2 contains character " << b2.getData()
90          << "\nObject d contains:\n" << d << "\n\n";
91     // visualizza i dati membro dell'oggetto della classe derivata
92     // l'operatore di risoluzione dello scope risolve l'ambiguità
93     // di getData
94     cout << "Data members of Derived can be"
95          << " accessed individually:"
96          << "\n     Integer: " << d.Base1::getData()
97          << "\n  Character: " << d.Base2::getData()
98          << "\nReal number: " << d.getReal() << "\n\n";
99
100    cout << "Derived can be treated as an "
101         << "object of either base class:\n";
102
103    // tratta Derived come oggetto Base1
104    base1Ptr = &d;
105    cout << "base1Ptr->getData() yields "
106         << base1Ptr->getData() << '\n';
107
108    // tratta Derived come oggetto Base2
109    base2Ptr = &d;
110    cout << "base2Ptr->getData() yields "
111         << base2Ptr->getData() << endl;
112
113    return 0;
114 }
```

```
Object b1 contains integer 10
Object b2 contains character Z
Object d contains:
     Integer: 7
   Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
     Integer: 7
   Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Figura 9.11 Esempio di ereditarietà multipla.

Osservate la semplice notazione dell'ereditarietà multipla, un segno di due punti dopo `class Derived` seguito dalla lista delle classi base separate da virgole. Notate inoltre che il costruttore di `Derived` chiama esplicitamente i costruttori delle due classi base, `Base1` e `Base2`, tramite la sintassi dell'inizializzatore di membro. I costruttori delle classi base sono chiamati nell'ordine specificato di ereditarietà, non nell'ordine in cui sono menzionati i loro costruttori. Inoltre se i costruttori delle classi base non sono chiamati esplicitamente nella lista degli inizializzatori di membro, ci sarà una chiamata implicita ai costruttori di default.

L'operatore di inserimento nello stream ridefinito da `Derived` utilizza la notazione puntata per accedere, tramite l'oggetto `d`, ai valori `value`, `letter` e `real` e per inviarli in output. Questa funzione operatore è `friend` di `Derived`, per cui può accedere direttamente a `real` che è un dato membro `private` di `Derived`. Inoltre, essendo `friend` di una classe derivata, può accedere ai membri protetti di `Base1` e `Base2`, rispettivamente `value` e `letter`.

Esaminiamo ora la funzione `main` del programma di esempio. Essa crea l'oggetto `b1` della classe `Base1` e lo inizializza al valore `10`. Crea poi l'oggetto `b2` della classe `Base2` e lo inizializza al valore `'Z'`. Infine crea l'oggetto `d` della classe `Derived` e lo inizializza in modo che contenga i valori `7`, `'A'` e `3.5`.

Il contenuto di ogni oggetto delle classi base viene visualizzato chiamando la funzione membro `getData` per ciascun oggetto. Anche se ci sono due funzioni `getData`, le chiamate non sono ambigue perché si riferiscono direttamente alle versioni dell'oggetto `b1` e dell'oggetto `b2`.

Successivamente il programma visualizza il contenuto di `d`, ma questa volta ci sono problemi di ambiguità, perché questo oggetto contiene due funzioni `getData`, una ereditata da `Base1` e l'altra da `Base2`. La risoluzione di questo problema è semplice se si usa l'operatore binario di risoluzione dello scope, scrivendo `d.Base1::getData()` per visualizzare `value` e `d.Base2::getData()` per visualizzare `letter` mentre il valore di `real` viene visualizzato senza ambiguità con `d.getReal()`. In seguito mostriamo che la relazione "is a" dell'ereditarietà singola vale anche per l'ereditarietà multipla. Assegniamo l'indirizzo dell'oggetto derivato `d` al puntatore alla classe base `base1Ptr` e visualizziamo il valore di `value` invocando la funzione membro di `Base1` `getData` tramite `base1Ptr`. Assegniamo poi l'indirizzo dell'oggetto derivato `d` al puntatore alla classe base `base2Ptr` e visualizziamo il valore di `letter` invocando la funzione membro di `Base2` `getData` tramite `base2Ptr`.

Questo è un semplice esempio che illustra il funzionamento dell'ereditarietà multipla e introduce il problema dell'ambiguità. L'ereditarietà multipla è un argomento complesso, trattato diffusamente nei testi sul C++ più avanzati.

Ingegneria del software 9.11

L'ereditarietà multipla offre grosse potenzialità, ma può introdurre complessità in un sistema: occorre progettare i sistemi con grande cura e dovrebbe essere evitata qualora l'ereditarietà singola fosse sufficiente.

9.16 Pensare in termini di oggetti : come sfruttare l'ereditarietà nel simulatore di ascensore [progetto opzionale]

In questo capitolo torniamo sul progetto del simulatore per capire se, e dove, sia possibile utilizzare i meccanismi di ereditarietà per semplificare la scrittura del codice o ottenere la riutilizzabilità di componenti. Fino ad ora abbiamo mantenuto distinte le classi `ElevatorButton` e `FloorButton`, ma in realtà esse hanno molte caratteristiche in comune, perché rappresentano entrambe un *tipo* di pulsante. Dunque esse sono delle ottime candidate per l'applicazione dell'ereditarietà. Per ottenere questo scopo dobbiamo individuare tali caratteristiche comuni e porle in una classe base che chiameremo `Button`. Da essa specializzeremo poi le classi `ElevatorButton` e `FloorButton`.

La domanda che dobbiamo porci è dunque quali siano le somiglianze tra `ElevatorButton` e `FloorButton`? La Figura 9.12 mostra gli attributi e le operazioni di entrambe le classi, così come sono stati dichiarati nei file di intestazione nel Capitolo 7 (rispettivamente nelle Figg. 7.24 e 7.26) e dal confronto risulta che le classi hanno in comune un attributo (`pressed`) e due operazioni (`pressButton` e `resetButton`). Scegliamo dunque di porre questi tre elementi nella classe base `Button`. Inoltre, nella nostra prima implementazione, `ElevatorButton` e `FloorButton` contengono ciascuna un riferimento a un oggetto della classe `Elevator` che possiamo ora spostare nella classe base `Button`.

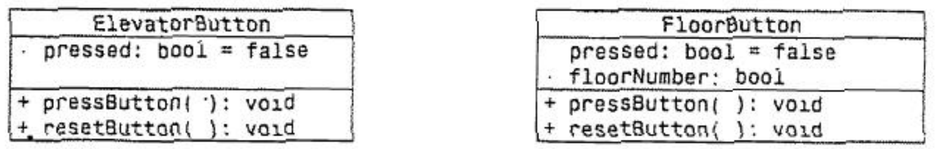


Figura 9.12 Attributi e operazioni delle classi ElevatorButton e FloorButton.

La Figura 9.13 mostra il nuovo progetto del simulatore. Notate che la classe `Floor` è composta da un oggetto della classe `FloorButton` e uno della classe `Light`, mentre la classe `Elevator` è composta da un oggetto della classe `ElevatorButton`, uno della classe `Door` e uno della classe `Bell`. Una linea continua con una freccia vuota collega ciascuna classe derivata alla classe base ed indica l'ereditarietà di `FloorButton` e di `ElevatorButton` da `Button`.

Resta ora una domanda: nelle classi derivate occorre effettuare l'overriding di qualche funzione della classe base? Se confrontiamo le funzioni membro `public` delle due classi (Figura 7.25 e 7.27) notiamo che la funzione membro `resetButton` è presente in entrambe ed è identica, per cui non c'è bisogno di overriding. La funzione `pressButton`, invece, è diversa nelle due classi: il codice di `pressButton` della classe `ElevatorButton` contiene le righe

```
pressed = true;
cout << "elevator button tells elevator to prepare to leave"
<< endl;
elevatorRef.prepareToLeave( true );
```

mentre la stessa funzione della classe `FloorButton` contiene le righe

```
pressed = true;
cout << "floor " << floorNumber
<< " button summons elevator" << endl;
elevatorRef.summonElevator( floorNumber );
```

I due blocchi di codice sono identici nella prima riga, ma si diversificano in seguito, per cui è necessario ridefinire la funzione `pressButton` nelle due sottoclassi.

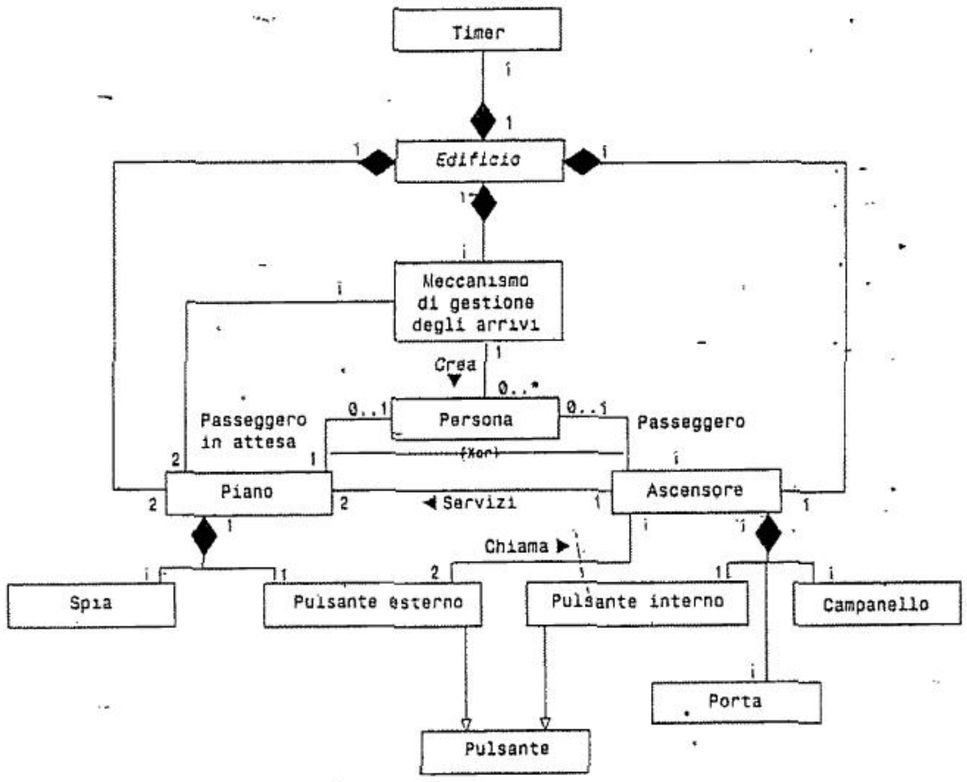


Figura 9.13 Diagramma delle classi dell'intero simulatore, con l'ereditarietà dalla classe Button.

La Figura 9.14 contiene il listato del file di intestazione della classe base `Button` [Si noti che, grazie all'incapsulamento, non dovremo cambiare altri file del simulatore. Le *uniche operazioni necessarie saranno quelle di sostituire i file di intestazione e di* implementazione di `elevatorButton` e `floorButton` con le nuove versioni, compilarli ed effettuare il linking dei file oggetto risultanti con quelli creati precedentemente e relativi agli altri file del simulatore]. Dichiariamo le funzioni membro `pressButton` e `resetButton` di visibilità `public` e il dato membro `pressed` di tipo `bool` e visibilità `private`. Notate la dichiarazione del riferimento alla classe `Elevator` alla linea 18 e il parametro corrispondente del costruttore alla linea 11. Vi mostreremo come inizializzare il riferimento quando analizzeremo il codice delle classi derivate.


```
1   // button.h
2   // Definizione della classe Button.
3   #ifndef BUTTON_H
4       #define BUTTON_H
5
6       class Elevator; // dichiarazione anticipata
7
8       class Button {
9
10          public:
11          Button( Elevator & ); // costruttore
12          ~Button(); // distruttore
13          void pressButton(); // pulsante premuto
14          void resetButton(); // pulsante rilasciato
15
16          protected:
17          // riferimento all'ascensore
18          Elevator &elevatorRef;
19
20          private:
21          bool pressed; // stato del pulsante
22      };
23
24  #endif // BUTTON_H
```

Figura 9.14 *File di intestazione della classe Button.*

Le classi derivate effettuano due operazioni diverse: `ElevatorButton` invoca la funzione membro `prepareToLeave` di `Elevator`, mentre `FloorButton` invoca la funzione membro `summonElevator`. Entrambe le classi, dunque, devono poter accedere al dato membro `elevatorRef` della classe base, ma ad esso non devono poter accedere oggetti di tipo diverso da `Button`, per questo motivo poniamo tale attributo nella sezione `protected` *della classe* `Button`. *Il dato membro* `pressed` *è dichiarato, invece, come* `private`, *in* quanto viene manipolato soltanto dalle funzioni membro della classe base, mentre quelle delle classi derivate non hanno bisogno di accedervi direttamente.

La Figura 9.15 contiene il file di implementazione della classe `Button`. La linea 9

```
: elevatorRef( elevatorHandle ), pressed( false )
```

inizializza il riferimento all'ascensore e assegna il valore `false` alla variabile che indica se il pulsante è premuto. Il costruttore e il distruttore visualizzano semplicemente dei messaggi che indicano la loro esecuzione, mentre le funzioni `pressButton` e `resetButton` manipolano il dato membro `pressed`.

In Figura 9.16 troviamo il listato del file di intestazione della classe `ElevatorButton`. Alla linea 10 specifichiamo che essa è una sottoclasse di `Button` e che eredita da quest'ultima con ereditarietà di tipo `public`. Ciò significa che `ElevatorButton` contiene il dato membro `protected` `elevatorRef` e le funzioni membro `public` della classe base `pressButton` e `resetButton`, e garantisce la stessa visibilità per queste componenti alle sue eventuali sottoclassi. Alla linea 15 dichiariamo il prototipo della funzione `pressButton`, segnalando il nostro intento di effettuarne l'overriding nel file `.cpp`. Parleremo tra un momento dell'implementazione di questa funzione.

```
1   // button.cpp
2   // Definizione delle funzioni membro della classe Button.
3   #include <iostream.h>
4
5   #include "button.h"
6
7   // costruttore
8   Button::Button( Elevator &elevatorHandle )
9   : elevatorRef( elevatorHandle ), pressed( false )
10  { cout << "button created" << endl; }
11
12  // distruttore
13  Button::~Button()
14  { cout << "button destroyed" << endl; }
15
16  // pulsante premuto
17  void Button::pressButton() { pressed = true; }
18
19  // pulsante azzerato
20  void Button::resetButton() { pressed = false; }
```

Figura 9.15 File di implementazione della classe Button.

```
1   // elevatorButton.h
2   // Definizione della classe ElevatorButton.
3   #ifndef ELEVATORBUTTON_H
4       #define ELEVATORBUTTON_H
5
6       #include "button.h"
7
8       class Elevator; // dichiarazione anticipata
9
10      class ElevatorButton : public Button {
11
12          public:
13          ElevatorButton( Elevator & ); // costruttore
14          ~ElevatorButton(); // distruttore
15          void pressButton(); // pulsante premuto
16      };
17
18  #endif // ELEVATORBUTTON_H
```

Figura 9.16 File di intestazione della classe ElevatorButton.

Il costruttore prende come parametro un riferimento alla classe `Elevator` (linea 13). Discuteremo della necessità di questo parametro quando analizzeremo il file di implementazione di questa classe. Notate, comunque, che abbiamo ancora bisogno della dichiarazione anticipata della classe `Elevator` (linea 8) per poter includere questo parametro nella dichiarazione del costruttore.

Il file di implementazione di `ElevatorButton` è presentato in Figura 9.17. I costruttori e i distruttori di questa classe visualizzano dei messaggi che indicano semplicemente la loro esecuzione. La linea 11:

```
: Button( elevatorHandle )
```

passa il riferimento ad `Elevator` al costruttore di classe base.

```
1   // elevatorButton.cpp:
2   // Definizione delle funzioni membro della classe ElevatorButton.
3   #include <iostream.h>
4
5   #include "elevatorButton.h"
6   #include "elevator.h"
7
8   // costruttore
9   ElevatorButton::ElevatorButton(
10  Elevator &elevatorHandle )
11  : Button( elevatorHandle )
12  { cout << "elevator button created" << endl; }
13
14  // distruttore
15  ElevatorButton::~ElevatorButton()
16  { cout << "elevator button destroyed" << endl; }
17
18  // pulsante premuto
19  void ElevatorButton::pressButton()
20  {
21      Button::pressButton();
22      cout << "elevator button tells elevator to prepare to leave"
23      << endl;
24      elevatorRef.prepareToLeave( true );
25  }
```

Figura 9.17 File di implementazione della classe ElevatorButton.

La ridefinizione della funzione `pressButton` chiama innanzitutto la sua omonima della classe base (linea 21), che ha come effetto quello di impostare l'attributo `pressed` al valore `true`. Successivamente, alla linea 24, si invoca la funzione `prepareToLeave` di `elevator` con l'argomento `true`, per indicare all'ascensore di andare all'altro piano.

La Figura 9.18 contiene il file di intestazione della classe `FloorButton`. Un'altra differenza tra questo file e quello della classe `ElevatorButton` è l'aggiunta del dato membro `floorNumber` alla linea 19, che sarà utilizzato per distinguere i due piani dell'edificio nei messaggi di output della simulazione. Nella dichiarazione del costruttore includiamo anche un parametro di tipo `int` (linea 13) in modo da poter inizializzare tale dato.

La Figura 9.19 riporta il file con l'implementazione della classe `FloorButton`. Alla linea 11 passiamo il riferimento ad `Elevator` al costruttore della classe base `Button` e inizializziamo il dato membro `floorNumber`. Il costruttore e il distruttore visualizzano i messaggi appropriati grazie al dato membro `floorNumber`. La ridefinizione della funzione

membro **pressButton** (linee 24-31) inizia chiamando la sua omonima della classe base, e invoca successivamente la funzione **summonElevator** di **elevator** passandogli **floorNumber**, cioè il piano da cui viene chiamato l'ascensore.

```
1   // floorButton.h
2   // Definizione della classe FloorButton.
3   #ifndef FLOORBUTTON_H
4       #define FLOORBUTTON_H
5
6       #include "button.h"
7
8       class Elevator; // dichiarazione anticipata
9
10      class FloorButton : public Button{
11
12          public:
13          FloorButton( int, Elevator & ); // costruttore
14          ~FloorButton(); // distruttore
15
16          void pressButton(); // pulsante premuto
17
18          private:
19          int floorNumber; // piano su cui si trova il pulsante
20      };
21
22  #endif // FLOORBUTTON_H
```

Figura 9.18 File di intestazione della classe FloorButton.

```
1   // floorButton.cpp
2   // Definizione delle funzioni membro della classe FloorButton.
3   #include <iostream.h>
4
5   #include "floorButton.h"
6   #include "elevator.h"
7
8   // costruttore
9   FloorButton::FloorButton( int number,
10  Elevator &elevatorHandle )
11  : Button( elevatorHandle ), floorNumber( number )
12  {
13      cout << "floor " << floorNumber << " button created"
14      << endl;
15  }
16
17  // distruttore
18  FloorButton::~FloorButton()
19  {
20     cout << "floor " << floorNumber << " button destroyed"
```

Figura 9.19 File di implementazione della classe FloorButton (continua)


```
21        << endl;
22  }
23
24  // pulsante premuto
25  void FloorButton::pressButton()
26  {
27      Button::pressButton();
28      cout << "floor " << floorNumber
29      << " button summons elevator" << endl;
30      elevatorRef.summonElevator( floorNumber );
31  }
```

Figura 9.19 File di implementazione della classe FloorButton.

A questo punto abbiamo completato l'implementazione del simulatore di ascensore, un progetto che ci ha accompagnati fino dal Capitolo 2. Naturalmente è ancora possibile introdurre miglioramenti strutturali al codice: per esempio, avrete notato che le classi **Button**, **Door** e **Light** hanno molte caratteristiche comuni, tra cui un attributo di "stato" e le operazioni di attivazione e disattivazione di tale attributo. E a pensarci bene, anche la classe **Bell** ha qualcosa in comune con altre classi del sistema. L'approccio orientato agli oggetti ci suggerirebbe di porre queste caratteristiche comuni in una o più classi base, dalle quali poi derivare le classi appropriate come abbiamo fatto in questo esempio con le classe **FloorButton** e **ElevatorButton**. Lasciamo come esercizio l'implementazione dell'ereditarietà per le classi **Button**, **Door**, **Light** e **Bell**. Vi suggeriamo di cominciare modificando il *diagramma delle classi in Figura 9.13. [Suggerimento:* **Button**, **Door** *e* **Light** *sono sostanzialmente* classi "interruttore", in quanto hanno uno *stato* e funzionalità di *attivazione* e *disattivazione*; **Bell** è una classe più semplice, con una sola operazione e senza stati.]

Speriamo davvero che lo studio del simulatore sia stata per voi un'esperienza di studio significativa e stimolante. Abbiamo adoperato un procedimento orientato agli-oggetti progressivo e pensato attentamente per presentare un progetto basato su UML. Partendo dal progetto abbiamo poi generato l'implementazione in C++ utilizzando le nozioni chiave della programmazione, tra cui classi, oggetti, incapsulamento, visibilità, composizione ed ereditarietà. Nei prossimi capitoli presenteremo altre funzionalità del linguaggio C++.

Esercizi di autovalutazione

9.1 Completate le seguenti affermazioni:

a) Se la classe **Alpha** deriva dalla classe Beta, la classe **Alpha** si chiama classe _____ e la classe **Beta** si chiama classe _____

b) In C++ esiste _____ che consente di derivare una classe da più classi base, anche se queste non sono tra di loro correlate.

c) *L'ereditarietà promuove il* _____ *che permette di risparmiare tempo nello sviluppo* del software e spinge a utilizzare software di alta qualità già collaudato.

d) Un oggetto di una classe _____ può essere trattato come un oggetto della sua classe _____

e) Per convertire un puntatore a una classe base in puntatore a una classe derivata, si deve effettuare un _____ perché il compilatore considera questa operazione rischiosa.

f) I tre specificatori di accesso ai membri sono _____, _____ e _____.

g) Quando si deriva una classe da una classe base con ereditarietà di tipo `public`, i membri `public` della classe base diventano membri _____ della classe derivata, e i membri `protected` della classe base diventano membri _____ della classe derivata.
h) Quando si deriva una classe da una classe base con ereditarietà di tipo `protected`, i membri `public` della classe base diventano membri _____ della classe derivata, e i membri `protected` della classe base diventano membri _____ della classe derivata.
i) La relazione "has a" tra classi rappresenta _____ e la relazione "is a" tra classi rappresenta _____.

Risposte agli esercizi di autovalutazione

9.1 a) derivata, base. b) ereditarietà multipla. c) riutilizzo del software. d) derivata, base. e) cast. f) `public`, `protected`, `private`. g) `public`, `protected`. h) `protected`, `protected`. i) composizione, ereditarietà.

Esercizi

9.2 Considerate la classe `Bicycle` (bicicletta). Con la vostra conoscenza delle comuni componenti di una bicicletta, create una gerarchia in cui la classe `Bicycle` deriva da altre classi, le quali derivano a loro volta da altre classi. Discutete l'istanza di vari oggetti della classe `Bicycle`. Discutete l'ereditarietà dalla classe `Bicycle` per altre classi strettamente correlate.

9.3 Definite brevemente i seguenti termini: ereditarietà, ereditarietà multipla, classe base e classe derivata.

9.4 Dite perché il compilatore considera rischiosa la conversione di un puntatore a una classe base in puntatore a una classe derivata.

9.5 Illustrate la differenza tra ereditarietà singola e multipla.

9.6 (Vero/Falso) Una classe derivata si dice anche sottoclasse perché rappresenta un sottoinsieme della classe base, vale a dire una classe derivata generalmente contiene meno oggetti della sua classe base.

9.7 (Vero/Falso) Un oggetto di una classe derivata è anche un oggetto della sua classe base.

9.8 Alcuni programmatori preferiscono non utilizzare la modalità di accesso `protected` perché viola l'incapsulamento della classe base. Mettete a confronto le modalità `protected` e `private` e discutetene pregi e difetti.

9.9 Molti programmi che fanno uso dell'ereditarietà potrebbero essere riscritti in termini di composizione e viceversa. Mettete a confronto i due approcci nel caso della gerarchia di `Point`, `Circle` e `Cylinder` che abbiamo incontrato in questo capitolo. Riscrivete il programma in Figura 9.10 (e le classi necessarie) utilizzando la composizione anziché l'ereditarietà. Infine, mettete nuovamente a confronto i due approcci e verificate la valutazione che ne avevate dato in un primo momento.

9.10 Riscrivete il programma di `Point`, `Circle` e `Cylinder` in Figura 9.10 come programma di `Point`, `Square` (quadrato) e `Cube` (cubo). Scrivete due versioni, una con l'ereditarietà e l'altra con la composizione.

9.11 Nel capitolo abbiamo affermato che quando un membro della classe base è inappropriato per la classe derivata, se ne può effettuare l'overriding, dandone un'implementazione appropriata. Se si fa ciò, sussiste ancora la relazione "is a" tra classe derivata e classe base? Spiegate la vostra risposta.

9.12 Studiate la gerarchia in Figura 9.2. Per ogni classe, indicate alcuni attributi e comportamenti in comune che giustificano la gerarchia. Aggiungete altre classi per arricchire la gerarchia.

9.13 Scrivete una gerarchia che mostri l'ereditarietà per le classi `Quadrilateral` (quadrilatero), `Trapezoid` (trapezio), `Parallelogram` (parallelogramma), `Rectangle` (rettangolo) e `Square` (quadrato). Utilizzate `Quadrilateral` come classe base della gerarchia. Rendete la gerarchia profonda il più possibile. I dati `private` di `Quadrilateral` dovrebbero essere le coppie di coordinate (`x`, `y`) dei quattro spigoli del `Quadrilateral`. Scrivete un programma di esempio che istanzia e visualizza gli oggetti di tutte queste classi.

9.14 Segnate su un foglio tutte le figure geometriche che vi vengono in mente, bidimensionali e tridimensionali, e datene una gerarchia di ereditarietà. La vostra gerarchia dovrebbe prevedere la classe base `Shape` (figura geometrica) da cui derivano le classi `TwoDimensionalShape` (figura bidimensionale) e `ThreeDimensionalShape` (figura tridimensionale). Dopo aver disegnato la gerarchia, definite ogni classe. Utilizzeremo questa gerarchia nel Capitolo 10 per vedere tutte le figure geometriche come oggetti della classe base `Shape`. Questa tecnica prende il nome di polimorfismo.

CAPITOLO 10

Le funzioni virtuali e il polimorfismo

Obiettivi

- Comprendere il concetto di polimorfismo
- Imparare a utilizzare le funzioni virtuali, che attuano il polimorfismo
- Comprendere le differenze che esistono tra classi astratte e classi concrete
- Imparare a dichiarare funzioni virtuali pure per creare classi astratte
- Comprendere l'importanza del polimorfismo nello sviluppo di software estensibile e di facile manutenzione
- Conoscere i dettagli implementativi delle funzioni virtuali e del binding dinamico

10.1 Introduzione

Grazie alle *funzioni virtuali* e al *polimorfismo* è possibile progettare e implementare facilmente sistemi *estensibili*. È possibile, infatti, trattare gli oggetti di tutte le classi di una gerarchia come se fossero oggetti della classe base. Le classi che non esistono ancora durante la stesura del programma, possono essere aggiunte alla gerarchia in seguito, effettuando alcune modifiche (poche o nessuna) alla parte del programma che opera sulle classi di base (generiche). Le uniche parti del programma che avranno bisogno di modifiche saranno quelle che richiedono una conoscenza specifica della nuova classe che si aggiunge alla gerarchia.

10.2 I campi di tipo e le istruzioni switch

Un modo di trattare tipi diversi consiste nell'utilizzare un'istruzione `switch`, che intraprende un'azione particolare a seconda del tipo dell'oggetto esaminato. Per esempio, in una gerarchia di forme geometriche in cui ogni forma specifica il suo tipo in un dato membro, un costrutto `switch` potrebbe determinare quale funzione `print` chiamare sulla base del tipo dell'oggetto esaminato.

L'uso di costrutti di tipo `switch` comporta, però, una serie di problemi: ad esempio potreste dimenticare di effettuare un test di tipo quando necessario oppure potreste dimenticare di includere nel costrutto `switch` tutti i possibili casi. Se è vero che una selezione `switch` si può sempre modificare aggiungendo nuovi tipi, potreste comunque dimenticare di inserire i nuovi casi in tutte gli `switch` esistenti. Ad ogni aggiunta o eliminazione

di una classe occorre modificare ogni costrutto `switch` e tenere questa complessità sotto controllo è una perdita di tempo e può generare degli errori. Come vedremo, le funzioni virtuali e i programmi polimorfi consentono di evitare gli inconvenienti insiti nella logica dello `switch`. Le funzioni virtuali implementano la stessa logica in modo automatico, scongiurando gli errori tipici associati ad esso.

Ingegneria del software 10.1

Utilizzando le funzioni virtuali e il polimorfismo semplificherete notevolmente l'aspetto dei vostri programmi: scompariranno le diramazioni logiche tipiche del costrutto `switch` e il codice sarà più semplice e sequenziale. Ciò è di grande aiuto nella messa a punto e nella manutenzione dei programmi, e consente di salvaguardarsi da un buon numero di errori.

10.3 Le funzioni virtuali

Supponiamo di avere un insieme di classi relative alle forme geometriche, come `Circle`, `Triangle`, `RightTriangle`, `Square`, e così via, che derivano dalla classe base `Shape`. In una buona progettazione ad oggetti, ognuna di esse dovrebbe essere dotata della capacità di visualizzare se stessa. Anche se ogni classe fosse dotata di una propria funzione `draw` (disegna), ogni versione di quest'ultima potrebbe essere anche sensibilmente diversa dalle altre. Indipendentemente dalla forma particolare che si vuole tracciare, sarebbe comodo poterla trattare genericamente come oggetto della classe base `Shape`. Per disegnare una qualsiasi forma, quindi, potrebbe essere sufficiente la chiamata alla funzione `draw` della classe base `Shape`, e dovrebbe essere poi il programma a determinare dinamicamente (cioè in fase di esecuzione) a quale classe derivata appartiene la funzione `draw` che si intende chiamare. Per ottenere un comportamento di questo tipo, possiamo dichiarare la funzione `draw` della classe base come *funzione virtuale* e effettuare la sua *ridefinizione* (*overriding*) in ogni classe derivata, in modo che disegni ogni diversa forma geometrica nel modo appropriato. Una funzione virtuale si dichiara precedendo il suo prototipo con la parola chiave `virtual` nella classe base. Per esempio nella classe base `Shape` potrebbe comparire la seguente definizione:

```
virtual void draw() const;
```

Questo prototipo dichiara `draw` come funzione virtuale costante che non prende argomenti e non restituisce alcun valore.

Ingegneria del software 10.2

Se dichiarate una funzione come virtuale, essa resta virtuale in tutta la gerarchia dell'ereditarietà da quel punto in poi, anche se non è dichiarata virtuale nella ridefinizione operata dalle altre classi.

Buona abitudine 10.1

Sebbene determinate funzioni siano implicitamente virtuali, a causa di una dichiarazione effettuata a un livello superiore nella gerarchia delle classi, è buona norma dichiararle esplicitamente come funzioni virtuali in ogni livello della gerarchia, per una maggiore chiarezza dei vostri programmi.

Ingegneria del software 10.3

Se in una classe derivata scegliete di non definire una funzione virtuale, tale classe eredita semplicemente la definizione della funzione presente nella sua classe base immediata (come avveniva normalmente fino-ad ora).

Supponiamo che la funzione `draw` della classe base sia dichiarata come virtuale: se utilizziamo un puntatore o un riferimento alla classe base per puntare all'oggetto della classe derivata e per invocare la funzione `draw` (per es., `shapePtr->draw()`), il programma sceglierà dinamicamente (durante l'esecuzione) la funzione `draw` della classe derivata appropriata. Ad esempio, se `shapePtr` si riferisce ad un cerchio la funzione `draw` disegnerà correttamente un cerchio. Questo comportamento si chiama *binding dinamico* e sarà illustrato in dettaglio nelle Sezioni 10.6 e 10.9.

Quando si chiama una funzione virtuale riferendosi ad un oggetto specifico tramite il nome e l'operatore punto (per es., `squareObject.draw()`), la risoluzione del riferimento avviene durante la compilazione (si ha il cosiddetto *binding statico*) e la funzione virtuale chiamata è quella definita, o ereditata, dalla classe dell'oggetto particolare. Nell'esempio, se `squareObject` si riferisce ad un quadrato la funzione `draw` che viene invocata su di esso disegnerà un quadrato. Tale chiamata viene determinata interamente in fase di compilazione.

10.4 Le classi base astratte e le classi concrete

Quando pensiamo a una classe come a un tipo di dato, diamo per scontato che saranno istanziati oggetti di quel tipo. Invece ci sono casi in cui serve definire delle classi per cui non si intende affatto istanziare oggetti. In questo caso si parla di *classi astratte*. Nell'ereditarietà, queste classi sono utilizzate come classi base, per cui normalmente prendono il nome di *classi base astratte*. Non è possibile istanziare alcun oggetto di una classe base astratta.

L'unico scopo di una classe astratta è diventare classe base di altre classi che ne erediteranno eventualmente interfaccia o l'implementazione mentre le classi utilizzabili per istanziare oggetti si chiamano *classi concrete*. Riprendendo l'esempio delle figure geometriche, possiamo pensare di avere una classe base astratta `TwoDimensionalShape` (figura geometrica bidimensionale) da cui derivare classi concrete come `Square`, `Circle`, `Triangle` e così via. Allo stesso modo possiamo prevedere una classe base astratta `ThreeDimensionalShape` (figura geometrica tridimensionale) da cui derivare classi concrete come `Cube`, `Sphere`, `Cylinder`, e così via. Le classi astratte sono troppo generiche perché possano definire oggetti reali; per farlo occorre essere più specifici e questo è proprio lo scopo delle classi concrete: fornire le informazioni specifiche che consentono di istanziare oggetti. Una classe si rende astratta dichiarando una o più funzioni virtuali come *pure*. Una *funzione virtuale pura* contiene l'inizializzatore `= 0` nella sua dichiarazione, come in

```
virtual float earnings() const = 0;   // virtuale pura
```

Ingegneria del software 10.4

Se derivate una classe da un'altra che contiene una funzione virtuale pura, e non fornite alcuna ridefinizione di tale funzione nella classe derivata, la funzione virtuale resterà pure anche nella classe derivata. Di conseguenza anche la classe derivata sarà una classe astratta.

Errore tipico 10.1

Se tentate di istanziare un oggetto di una classe astratta (cioè di una classe che contiene almeno una funzione virtuale pura) commettete un errore di sintassi.

Una gerarchia non deve necessariamente contenere classi astratte, ma normalmente in fase di progettazione si definiscono delle classi astratte nei primi livelli della gerarchia. Un esempio di questo è dato dalla gerarchia delle forme geometriche che potrebbe partire con la classe base astratta `Shape`. Nel livello successivo potremmo porre ancora due classi astratte, cioè `TwoDimensionalShape` e `ThreeDimensionalShape` mentre al terzo livello possiamo, invece, cominciare a definire classi concrete per le forme bidimensionali (ad es. cerchi e rettangoli) e per quelle tridimensionali (ad es. sfere e cubi).

10.5 Il polimorfismo

Il *polimorfismo* è la capacità che hanno oggetti di classi diverse ma correlate per il fatto di derivare da una classe base comune di rispondere in maniera diversa a uno stesso messaggio. Lo stesso messaggio inviato a diversi tipi di oggetti assume quindi diverse "forme", da cui il termine polimorfismo. Per esempio, se la classe `Rectangle` deriva dalla classe `Quadrilateral`, un oggetto `Rectangle` è una versione più specifica di un oggetto `Quadrilateral`. Un'operazione che ha senso su un oggetto `Quadrilateral`, come il calcolo del perimetro o dell'area, si può effettuare anche su un oggetto `Rectangle` ma su di esso avrà "forma" (cioè comportamento) diverso.

Il polimorfismo si implementa tramite le funzioni virtuali. Alla richiesta di utilizzare una funzione virtuale tramite un puntatore o un riferimento alla classe base, il C++ risponde scegliendo la ridefinizione corretta della funzione nella classe derivata appropriata associata all'oggetto.

A volte si definisce una funzione membro non virtuale in una classe base e se ne effettua l'overriding in una classe derivata. Se questa funzione viene chiamata tramite un puntatore alla classe base che punta a un oggetto della classe derivata, viene utilizzata la versione della classe base mentre se viene chiamata tramite un puntatore alla classe derivata, viene utilizzata la versione della classe derivata; in questo caso non dà luogo ad un comportamento "polimorfo".

Consideriamo questo esempio che riguarda la classe base `Employee` e la classe derivata `HourlyWorker` in Figura 9.5:

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();      // chiama la funzione print della classe base
hPtr->print();      // chiama la funzione print della classe derivata
ePtr = &h;          // conversione implicita consentita
ePtr->print();      // chiama anch'essa la funzione print della classe
                    // base
```

La classe base `Employee` e la classe derivata `HourlyWorker` definiscono entrambe le proprie funzioni `print`. Dal momento che le due funzioni non sono state dichiarate come virtuali e hanno segnatura uguale, se si chiama `print` tramite un puntatore di tipo `Employee` si chiama in realtà `Employee::print()`, indipendentemente dal fatto che il puntatore

punti a un oggetto `Employee` o `HourlyWorker`, mentre se si chiama `print` tramite un puntatore di tipo `HourlyWorker` si chiama `HourlyWorker::print()`. La versione di `print` della classe base è disponibile anche alla classe derivata, ma per chiamare tale versione di `print` per un oggetto di una classe derivata tramite un puntatore a un oggetto della classe derivata, la funzione chiamata deve essere esplicitata come segue:

```
hPtr->Employee::print();  // chiama la funzione print della classe
                          // base
```

In questo modo si specifica esplicitamente che va chiamata la funzione `print` della classe base.

Grazie al polimorfismo e alle funzioni virtuali, la chiamata a una funzione membro può avere effetti diversi, a seconda del tipo dell'oggetto che riceve la chiamata, anche se vedremo che questo fatto incide sia pure in lieve misura sull'efficienza dell'esecuzione. Le capacità espressive del codice risultano notevolmente ampliate. Nelle prossime sezioni esploreremo le potenzialità del polimorfismo e delle funzioni virtuali grazie a una serie di esempi significativi.

Ingegneria del software 10.5

Grazie al polimorfismo e alle funzioni virtuali, il programmatore può trattare un problema in modo generale, delegando i dettagli specifici all'ambiente di esecuzione. Il programmatore può avere il controllo di una vasta gamma di oggetti senza neanche conoscerne il tipo.

Ingegneria del software 10.6

Il polimorfismo promuove l'estensibilità del software: i programmi che invocano comportamenti polimorfi sono indipendenti dal tipo di oggetti a cui inviano i messaggi. In questo modo, si possono aggiungere al sistema originario nuovi tipi di oggetto che rispondono ai messaggi esistenti senza dover effettuare modifiche sul codice già scritto. Eccetto che per i client che istanziano nuovi oggetti, i programmi non devono neppure essere ricompilati.

Ingegneria del software 10.7

Una classe astratta definisce un'interfaccia per i vari membri di una gerarchia di classi: essa contiene funzioni virtuali pure che saranno definite nelle classi derivate. Grazie al polimorfismo tutte le funzioni della gerarchia possono utilizzare la stessa interfaccia.

Anche se non è possibile istanziare oggetti di una classe base astratta, è possibile dichiarare puntatori e riferimenti a oggetti di tale classe. Essi possono essere utili nelle manipolazioni polimorfe degli oggetti delle classi derivate concrete.

Vediamo quali applicazioni possono avere il polimorfismo e le funzioni virtuali. Uno "screen manager" è un programma che si occupa della gestione dello schermo e deve visualizzare una varietà di oggetti di classi diverse, tra cui nuovi tipi di oggetto che saranno aggiunti al sistema anche dopo la sua scrittura. Il sistema potrebbe dover visualizzare varie forme geometriche (di classe base `Shape`) come quadrati, cerchi, triangoli, rettangoli, punti, linee e simili. Lo screen manager usa puntatori e riferimenti alla classe base (`Shape`) per gestire tutti gli oggetti da visualizzare; per disegnare un oggetto lo screen manager usa un puntatore o un riferimento all'oggetto del tipo della classe base a cui invia semplicemen-

te il messaggio `draw`. La funzione `draw` è dichiarata come funzione virtuale pura nella classe `Shape` ed è ridefinita in ogni classe derivata in modo che ogni oggetto `Shape` sappia come disegnare se stesso (questo meccanismo è anche chiamato *principio del controllo invertito* poiché è lo screen manager che chiede e delega agli oggetti di disegnarsi e non viceversa). Lo screen manager non deve preoccuparsi di conoscere di che oggetto si tratta o se è un oggetto che ha già incontrato in precedenza: l'unica cosa che fa è dire all'oggetto di disegnare se stesso.

Il polimorfismo sfodera tutta la sua potenzialità nell'implementazione di sistemi software costituiti da più livelli logici. Ad esempio, nel caso dei sistemi operativi ogni periferica può operare in modo diverso dalle altre anche se i vari comandi per la lettura e la scrittura dei dati dalverso le periferiche possono avere una certa uniformità. Il messaggio *write* (scrittura) inviato a un oggetto driver di una periferica deve essere interpretato specificatamente nel contesto di quel driver e del modo in cui esso gestisce la periferica. Comunque una *write* specifica non è diversa da altre *write* inviate ad altre periferiche del sistema: tutto ciò che fa è trasferire un determinato numeró di byte dalla memoria alla periferica. Nella definizione di un sistema operativo orientato agli oggetti, si potrebbe dunque utilizzare una classe base astratta per l'interfaccia di tutti i driver di periferica e, grazie all'ereditarietà, si potrebbero derivare direttamente delle classi che opereranno in modo simile. Le funzionalità (l'interfaccia pubblica) fornite dai driver di periferica sono in ultima analisi funzioni virtuali pure della classe base astratta e le loro implementazioni sono fornite nelle classi derivate, che corrispondono agli specifici driver di periferica.

Con il polimorfismo, un programma può attraversare un array di puntatori a oggetti che appartengono a vari livelli di una gerarchia di classi. I puntatori di un array di questo tipo saranno tutti puntatori del tipo della classe base che puntano ad istanze concrete di oggetti delle classi derivate. Per esempio, un array di oggetti `TwoDimensionalShape` potrebbe contenere puntatori di tipo `TwoDimensionalShape *` a oggetti delle classi derivate `Square`, `Circle`, `Triangle`, `Rectangle`, `Line` e così via. Per disegnare ciascuno di questi oggetti basta inviargli un messaggio `draw`.

10.6 Progettazione di un libro paga elettronico

Adesso vogliamo mostrarvi come creare un libro paga elettronico per diversi tipi di dipendente utilizzando il polimorfismo e le funzioni virtuali (Figura 10.1) definite nella classe base `Employee`. Le classi derivate di `Employee` sono `Boss`, che prende un fisso settimanale indipendentemente dal numero di ore di presenza, `CommissionWorker` che prende un fisso iniziale più una percentuale sulle vendite, `PieceWorker` che viene pagato sul numero di prodotti lavorati e `HourlyWorker` che prende un fisso orario per le ore ordinarie e un fisso maggiorato per le ore di straordinario.

La funzione `earnings` (guadagno) si potrebbe certamente applicare in modo generico a tutti gli impiegati. Tuttavia il modo in cui ogni impiegato guadagna il proprio stipendio varia a seconda della classe a cui egli appartiene, e tutte le classi derivano dalla classe base `Employee`. Perciò `earnings` è dichiarata come funzione virtuale pura nella classe base `Employee`, mentre ogni classe derivata ne dà una sua implementazione. Per calcolare il guadagno di qualsiasi impiegato, quindi, il programma utilizza semplicemente un puntatore o un riferimento del tipo della classe base per un dato oggetto impiegato e invoca la funzione `earnings`. In un vero libro paga elettronico, avremmo un array o una lista di

puntatori di tipo `Employee *`, in cui ciascun elemento punterebbe a un oggetto impiegato. Il programma dovrebbe semplicemente scandire l'array un elemento alla volta (utilizzando puntatori di tipo `Employee *`) per invocare la funzione `earnings` di ciascun oggetto.

```
1   // Figura 10.1: employ2.h
2   // La classe base astratta Employee
3   #ifndef EMPLOY2_H
4   #define EMPLOY2_H
5
6   #include <iostream.h>
7
8   class Employee {
9   public:
10     Employee( const char *, const char * );
11     ~Employee();   // distruttore
12     const char *getFirstName() const;
13     const char *getLastName() const;
14
15  // Una funzione virtuale pura rende Employee classe base astratta
16     virtual double earnings() const = 0;   // virtuale pura
17     virtual void print() const;            // virtuale
18  private:
19     char *firstName;
20     char *lastName;
21  };
22
23  #endif
24  // Figura 10.1: employ2.cpp
25  // Definizioni delle funzioni membro della
26  // classe base astratta Employee.
27  // Nota: Nessuna definizione per le funzioni virtuali pure.
28  #include <string.h>
29  #include <assert.h>
30  #include "employ2.h"
31
32  // Il costruttore alloca dinamicamente spazio per
33  // il nome e il cognome e utilizza strcpy per copiarli
34  // nell'oggetto.
35  Employee::Employee( const char *first, const char *last )
36  {
37     firstName = new char[ strlen( first ) + 1 ];
38     assert( firstName != 0 );     // new ha funzionato?
39     strcpy( firstName, first );
40
41     lastName = new char[ strlen( last ) + 1 ];
42     assert( lastName != 0 );      // new ha funzionato?
43     strcpy( lastName, last );
44  }
45
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)

```
46  // Il distruttore libera la memoria allocata dinamicamente
47  Employee::~Employee()
48  {
49     delete [] firstName;
50     delete [] lastName;
51  }
52
53  // Restituisce un puntatore al nome. Il tipo restituito è const
54  // per evitare che il chiamante possa modificare i dati private.
55  // Il chiamante dovrebbe copiare la stringa restituita prima che
56  // il distruttore effettui la delete sulla memoria dinamica, per
57  // evitare di operare con un puntatore indefinito.
58  const char *Employee::getFirstName() const
59  {
60     return firstName;   // il chiamante deve copiare la stringa
61  }
62
63  // Restituisce un puntatore al cognome
64  // Il tipo restituito è const per evitare che il chiamante possa
65  // modificare i dati private. Il chiamante dovrebbe copiare la
66  // stringa restituita prima che il distruttore effettui la delete
67  // sulla memoria dinamica, per evitare un puntatore indefinito.
68  const char *Employee::getLastName() const
69  {
70     return lastName;   // il chiamante deve copiare la stringa
71  }
72
73  // Visualizza il nome dell'impiegato
74  void Employee::print() const
75     { cout << firstName << ' ' << lastName; }
76  // Figura 10.1: boss1.h
77  // Classe Boss derivata da Employee
78  #ifndef BOSS1_H
79  #define BOSS1_H
80  #include "employ2.h"
81
82  class Boss : public Employee {
83  public:
84     Boss( const char *, const char *, double = 0.0 );
85     void setWeeklySalary( double );
86     virtual double earnings() const;
87     virtual void print() const;
88  private:
89     double weeklySalary;
90  };
91
92  #endif
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)


```
93  // Figura 10.1: boss1.cpp
94  // Definizioni delle funzioni membro della classe Boss
95  #include "boss1.h"
96
97  // Costruttore di Boss
98  Boss::Boss( const char *first, const char *last, double s )
99     : Employee( first, last )  // chiama il costruttore della
100                               // classe base
101 { setWeeklySalary( s ); }
102
103 // Imposta la paga del Boss
104 void Boss::setWeeklySalary( double s )
105    { weeklySalary = s > 0 ? s : 0; }
106
107 // Restituisce la paga del Boss
108 double Boss::earnings() const { return weeklySalary; }
109
110 // Visualizza il nome del Boss
111 void Boss::print() const
112 {
113    cout << "\n            Boss: ";
114    Employee::print();
115 }
116 // Figura 10.1: commis1.h
117 // Classe CommissionWorker derivata da Employee
118 #ifndef COMMIS1_H
119 #define COMMIS1_H
120 #include "employ2.h"
121
122 class CommissionWorker : public Employee {
123 public:
124    CommissionWorker( const char *, const char *,
125                      double = 0.0, double = 0.0,
126                      int = 0 );
127    void setSalary( double );
128    void setCommission( double );
129    void setQuantity( int );
130    virtual double earnings() const;
131    virtual void print() const;
132 private:
133    double salary;        // paga base settimanale
134    double commission;    // somma spettante per unità di prodotto
135                          // venduta
136    int quantity;     // totale delle unità vendute nella settimana
137 };
138
139 #endif
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)

```
// Figura 10.1: commis1.cpp - Definizioni delle
// funzioni membro della classe CommissionWorker
#include <iostream.h>
#include "commis1.h"

// Costruttore di CommissionWorker
CommissionWorker::CommissionWorker( const char *first,
      const char *last, double s, double c, int q )
   : Employee( first, last )  // chiama il costruttore della
                              // classe base
{
   setSalary( s );
   setCommission( c );
   setQuantity( q );
}

// Imposta la paga base settimanale di CommissionWorker
void CommissionWorker::setSalary( double s )
   { salary = s > 0 ? s : 0; }

// Imposta la commissione di CommissionWorker
void CommissionWorker::setCommission( double c )
   { commission = c > 0 ? c : 0; }

// Imposta la quantità di prodotto venduto da CommissionWorker
void CommissionWorker::setQuantity( int q )
   { quantity = q > 0 ? q : 0; }

// Determina il guadagno di CommissionWorker
double CommissionWorker::earnings() const
   { return salary + commission * quantity; }

// Visualizza il nome di CommissionWorker
void CommissionWorker::print() const
{
   cout << "\nCommission worker: ";
   Employee::print();
}
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)

Osserviamo la classe `Employee` (employ2.h ed employ2.cpp). Le funzioni membro `public` includono un costruttore che prende come argomenti il nome e il cognome dell'impiegato, un distruttore che restituisce al sistema la memoria allocata dinamicamente, una funzione *get* che restituisce il nome dell'impiegato, una funzione *get* che restituisce il cognome dell'impiegato, la funzione virtuale pura `earnings` e la funzione virtuale `print`. Potremmo chiederci i motivi per cui `earnings` è definita come funzione virtuale pura. La risposta è che non ha senso scriverne una versione per la classe base: non possiamo calcolare il guadagno di un impiegato generico ma dobbiamo prima sapere di che tipo di impiegato si tratti. Rendendo questa funzione virtuale pura indichiamo che ne scriveremo

l'implementazione in ciascuna classe derivata. Non avremo mai intenzione di chiamare questa funzione virtuale pura nella classe base astratta `Employee`: tutte le classi derivate effettueranno la ridefinizione di `earnings` fornendone la propria implementazione.

La classe `Boss` (boss1.h e boss1.cpp) deriva da `Employee` con ereditarietà di tipo `public`. Le funzioni membro `public` includono un costruttore che prende come argomenti il nome, il cognome e li passa il al costruttore di `Employee` per inizializzare i membri `firstName` e `lastName` della classe base; il terzo argomento, invece costituisce la paga settimanale. Troviamo poi una funzione *set* che assegna un nuovo valore al dato membro `private weeklySalary` e la funzione virtuale `earnings` che definisce come calcolare il guadagno di un `Boss`. Infine, la funzione virtuale `print` visualizza il tipo dell'impiegato e chiama poi `Employee::print()` per visualizzarne il nome.

La classe `CommissionWorker` (commis1.h e commis1.cpp) deriva da `Employee` con ereditarietà di tipo `public`. Le funzioni membro `public` includono un costruttore che prende come argomenti il nome, il cognome, la paga base, la percentuale di commissione e il numero di prodotti venduti, e passa il nome e il cognome al costruttore di `Employee`. Troviamo poi le funzioni *set* che assegnano nuovi valori ai dati membro `private salary`, `commission` e `quantity`, la funzione virtuale `earnings` che definisce come calcolare i guadagni di un `CommissionWorker` e la funzione virtuale `print` che visualizza il tipo di impiegato e chiama `Employee::print()` per visualizzarne il nome.

La classe `PieceWorker` (piece1.h e piece1.cpp) deriva da `Employee` con ereditarietà di tipo `public`. Le funzioni membro `public` includono un costruttore che prende come argomenti il nome, il cognome, il salario per unità di prodotto e il numero di unità prodotte, e passa il nome e il cognome al costruttore di `Employee`. Troviamo poi le funzioni *set* che assegnano nuovi valori ai dati membro `private wagePerPiece` e `quantity`, la funzione virtuale `earnings` che definisce come calcolare i guadagni di un `PieceWorker` e la funzione virtuale `print` che visualizza il tipo di impiegato e chiama `Employee::print()` per visualizzarne il nome.

```
// Figura 10.1: piece1.h
// Classe PieceWorker derivata da Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
   PieceWorker( const char *, const char *,
                double = 0.0, int = 0);
   void setWage( double );
   void setQuantity( int );
   virtual double earnings() const;
   virtual void print() const;
private:
   double wagePerPiece; // salario per unità di prodotto
   int quantity; // quantità di prodotti lavorati nella settimana
};
#endif
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)

```
196 // Figura 10.1: piece1.cpp
197 // Definizioni delle funzioni membro della classe PieceWorker
198 #include <iostream.h>
199 #include "piece1.h"
200
201 // Costruttore di PieceWorker
202 PieceWorker::PieceWorker( const char *first, const char *last,
203                           double w, int q )
204    : Employee( first, last )  // chiama il costruttore della
205                               // classe base
206 {
207    setWage( w );
208    setQuantity( q );
209 }
210
211 // Imposta il salario unitario
212 void PieceWorker::setWage( double w )
213    { wagePerPiece = w > 0 ? w : 0; }
214
215 // Imposta il numero di prodotti lavorati
216 void PieceWorker::setQuantity( int q )
217    { quantity = q > 0 ? q : 0; }
218
219 // Determina il guadagno di PieceWorker
220 double PieceWorker::earnings() const
221    { return quantity * wagePerPiece; }
222
223 // Visualizza il nome di PieceWorker
224 void PieceWorker::print() const
225 {
226    cout << "\n     Piece worker: ";
227    Employee::print();
228 }
229 // Figura 10.1: hourly1.h
230 // Definizione della classe HourlyWorker
231 #ifndef HOURLY1_H
232 #define HOURLY1_H
233 #include "employ2.h"
234
235 class HourlyWorker : public Employee {
236 public:
237    HourlyWorker( const char *, const char *,
238                  double = 0.0, double = 0.0);
239    void setWage( double );
240    void setHours( double );
241    virtual double earnings() const;
242    virtual void print() const;
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)


```
243 private:
244    double wage;   // paga oraria
245    double hours;  // ore di lavoro della settimana
246 };
247
248 #endif
249 // Figura 10.1: hourly1.cpp
250 // Definizioni delle funzioni membro della classe HourlyWorker
251 #include <iostream.h>
252 #include "hourly1.h"
253
254 // Costruttore di HourlyWorker
255 HourlyWorker::HourlyWorker( const char *first,
256                             const char *last,
257                             double w, double h )
258    : Employee( first, last )   // chiama il costruttore della
259                                // classe base
260 {
261    setWage( w );
262    setHours( h );
263 }
264
265 // Imposta la paga oraria
266 void HourlyWorker::setWage( double w )
267    { wage = w > 0 ? w : 0; }
268
269 // Imposta le ore di lavoro effettuate
270 void HourlyWorker::setHours( double h )
271    { hours = h >= 0 && h < 168 ? h : 0; }
272
273 // Restituisce la paga di HourlyWorker
274 double HourlyWorker::earnings() const
275 {
276    if ( hours <= 40 ) // niente straordinario
277       return wage * hours;
278    else               // lo straordinario è pagato con una
279                       // maggiorazione del 50%
280       return 40 * wage + ( hours - 40 ) * wage * 1.5;
281 }
282
283 // Visualizza il nome di HourlyWorker
284
285 void HourlyWorker::print() const
286 {
287    cout << "\n    Hourly worker: ";
288    Employee::print();
289 }
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi `Employee` (continua)

La classe **HourlyWorker** (hourly1.h e hourly1.cpp) deriva da **Employee** con ereditarietà di tipo **public**. Le funzioni membro **public** includono un costruttore che prende come argomenti il nome, il cognome, il salario orario e il numero di ore lavorate, e passa il nome e il cognome al costruttore di **Employee**. Troviamo poi le funzioni *set* che assegnano nuovi valori ai dati membro **private** **wage** e **hours**, la funzione virtuale **earnings** che definisce come calcolare i guadagni di un **HourlyWorker** e la funzione virtuale **print** che visualizza il tipo di impiegato e chiama **Employee::print()** per visualizzarne il nome.

Il programma di esempio e i suoi output sono presentati in fig10_01.cpp. I quattro segmenti di codice in **main** sono simili, per cui ne discuteremo soltanto il primo relativo a un oggetto **Boss**.

```
290  // Figura 10.1: fig10_01.cpp
292  // Programma di esempio per la gerarchia Employee
293  #include <iostream.h>
294  #include <iomanip.h>
295  #include "employ2.h"
296  #include "boss1.h"
297  #include "commis1.h"
298  #include "piece1.h"
299  #include "hourly1.h"
300
301  void virtualViaPointer( const Employee * );
302  void virtualViaReference( const Employee & );
303
304  int main()
305  {
306     // imposta la formattazione dell'output
307     cout << setiosflags( ios::fixed | ios::showpoint )
308          << setprecision( 2 );
309
310     Boss b( "John", "Smith", 800.00 );
311     b.print();                                  // binding statico
312     cout << " earned $" << b.earnings();       // binding statico
313     virtualViaPointer( &b );                   // utilizza il binding dinamico
314     virtualViaReference( b );                  // utilizza il binding dinamico
315
316     CommissionWorker c( "Sue", "Jones", 200.0, 3.0, 150 );
317     c.print();                                  // binding statico
318     cout << " earned $" << c.earnings();       // binding statico
319     virtualViaPointer( &c );                   // utilizza il binding dinamico
320     virtualViaReference( c );                  // utilizza il binding dinamico
321
322     PieceWorker p( "Bob", "Lewis", 2.5, 200 );
323     p.print();                                  // binding statico
324     cout << " earned $" << p.earnings();       // binding statico
325     virtualViaPointer( &p );                   // utilizza il binding dinamico
326     virtualViaReference( p );                  // utilizza il binding dinamico
327
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi **Employee** (continua).


```
328     HourlyWorker h( "Karen", "Price", 13.75, 40 );
329     h.print();                                  // binding statico
330     cout << " earned $" << h.earnings();       // binding statico
331     virtualViaPointer( &h );                   // utilizza il binding dinamico
332     virtualViaReference( h );                  // utilizza il binding dinamico
333     cout << endl;
334     return 0;
335  }
336
337  // Chiamate a funzioni virtuali effettuate tramite un puntatore
338  // di classe base utilizzando il binding dinamico.
339  void virtualViaPointer( const Employee *baseClassPtr )
340  {
341     baseClassPtr->print();
342     cout << " earned $" << baseClassPtr->earnings();
343  }
344
345  // Chiamate a funzioni virtuali effettuate tramite un riferimento
346  // di classe base utilizzando il binding dinamico.
347  void virtualViaReference( const Employee &baseClassRef )
348  {
349     baseClassRef.print();
350     cout << " earned $" << baseClassRef.earnings();
351  }
```

```
Boss: John Smith earned $800.00
Boss: John Smith earned $800.00
Boss: John Smith earned $800.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Piece worker: Bob Lewis earned $500.00
Piece worker: Bob Lewis earned $500.00
Piece worker: Bob Lewis earned $500.00
Hourly worker: Karen Price earned $550.00
Hourly worker: Karen Price earned $550.00
Hourly worker: Karen Price earned $550.00
```

Figura 10.1 Implementazione polimorfica della gerarchia di classi **Employee**.

La linea 310

```
Boss b( "John", "Smith", 800.00 );
```

istanzia l'oggetto **b** della classe derivata **Boss**, fornendo al costruttore gli argomenti nome, cognome e fisso settimanale.

La linea 311

```
b.print();                                  // binding statico
```

invoca esplicitamente la funzione `print` nella *versione di* `Boss` *tramite l'operatore di selezione* punto. Questo è un esempio di binding statico perché il tipo di oggetto per cui viene chiamata la funzione è noto già durante la compilazione. La chiamata è presente a scopo comparativo, per illustrare che il binding dinamico invoca la giusta funzione `print`.

La linea 312

```
cout << " earned $" << b.earnings();     // binding statico
```

invoca esplicitamente la funzione `earnings` nella versione di `Boss` *tramite l'operatore* punto. È un altro esempio di binding statico presentato a scopo puramente comparativo.

La linea 313

```
virtualViaPointer( &b );              // utilizza il binding dinamico
```

invoca la funzione `virtualViaPointer` (linea 331) con l'indirizzo dell'oggetto `b` della classe derivata. La funzione riceve questo indirizzo nel parametro `baseClassPtr` che è dichiarato come `const Employee *`. Questo è esattamente il modo in cui implementare un comportamento polimorfo.

La linea 341

```
baseClassPtr->print();
```

invoca la funzione membro `print` dell'oggetto a cui punta `baseClassPtr`. Dato che `print` è dichiarata come funzione virtuale nella classe base, il sistema invoca la funzione `print` dell'oggetto della classe derivata, ancora una volta otteniamo un comportamento polimorfo. Questa chiamata di funzione è un esempio di binding dinamico, perché la funzione virtuale è invocata tramite un puntatore del tipo della classe base, per cui la decisione su quale funzione chiamare è rimandata alla fase di esecuzione.

La linea 342

```
cout << " earned $" << baseClassPtr->earnings();
```

invoca la funzione membro `earnings` dell'oggetto a cui punta `baseClassPtr`. Dato che `earnings` è dichiarata come funzione virtuale nella classe base, il sistema invoca la funzione `earnings` dell'oggetto della classe derivata. Anche questo è un esempio di binding dinamico.

La linea 314

```
virtualViaReference( b );             // utilizza il binding dinamico
```

invoca la funzione `virtualViaReference` (linea 347) per mostrare come ottenere il polimorfismo chiamando le funzioni virtuali tramite riferimenti alla classe base. La funzione riceve l'oggetto `b` nel parametro `baseClassRef` che è dichiarato come `const Employee &`.

La linea 349

```
baseClassRef.print();
```

invoca la funzione membro `print` dell'oggetto referenziato da `baseClassRef`. Dal momento che `print` è dichiarata come funzione virtuale nella classe base, il sistema invoca la funzione `print` dell'oggetto della classe derivata. Anche questa chiamata di funzione è un esempio di binding dinamico, perché la funzione viene invocata tramite un riferimento alla classe base, per cui la decisione su quale funzione invocare è rimandata alla fase di esecuzione.

La linea 350

```
cout << " earned $" << baseClassRef.earnings();
```

invoca la funzione membro `earnings` dell'oggetto riferito da `baseClassRef`. Dato che `earnings` *è dichiarato come funzione virtuale nella classe base, il sistema invoca la funzione* `earnings` dell'oggetto della classe derivata: è un altro esempio di binding dinamico.

10.7 L'aggiunta di nuove classi e il binding dinamico

Il polimorfismo e le funzioni virtuali operano abbastanza bene se tutte le classi possibili sono note in anticipo ma funzionano anche quando nuove classi si aggiungono al sistema e vengono "accolte" dal binding dinamico (detto anche *late binding* o *binding differito*). Il tipo di un oggetto non deve necessariamente essere noto durante la compilazione, perché *una chiamata a una funzione virtuale possa essere compilata correttamente ma è durante* l'esecuzione che la funzione virtuale viene sostituita con la funzione appropriata.

Uno screen manager può quindi visualizzare nuovi tipi di oggetti, man mano che *vengono aggiunti al sistema, senza dover essere ricompilato ogni volta. La funzione* `draw` resta uguale a se stessa: saranno i nuovi oggetti a contenere le vere funzionalità di disegno. Ciò semplifica l'aggiunta di nuove funzionalità, riducendo al minimo l'impatto che hanno sul sistema e costituisce un punto cruciale per il riutilizzo del software.

Il binding dinamico consente ai produttori di distribuire il proprio software senza rivelarne l'implementazione. La distribuzione del software si limita alla vendita di file di intestazione e di file oggetto senza divulgare il codice sorgente. Chi scrive software può utilizzare l'ereditarietà per derivare nuove classi da quelle fornite dai produttori ed il software che funziona con queste continuerà a funzionare con le classi derivate e, grazie al binding dinamico, utilizzerà le funzioni virtuali ridefinite nelle nuove classi.

Nella Sezione 10.9 presentiamo un altro esempio completo di polimorfismo. Nella Sezione 10.10 descriviamo in modo approfondito come sono implementati in C++ il polimorfismo, le funzioni virtuali e il binding dinamico.

10.8 I distruttori virtuali

Quando si utilizza il polimorfismo per trattare oggetti di una gerarchia di classi allocati dinamicamente può verificarsi un problema. Se un oggetto viene distrutto esplicitamente applicando l'operatore `delete` a un puntatore all'oggetto del tipo della classe base, viene invocata la funzione distruttore della classe base. Ciò si verifica indipendentemente dal *tipo di oggetto a cui esso punta e dal fatto che il distruttore di ogni classe abbia un nome* diverso.

La soluzione a questo problema è semplice: basta dichiarare un distruttore virtuale *nella classe base. Ciò rende automaticamente virtuali tutti i distruttori delle classi derivate,* anche se non hanno lo stesso nome di quello della classe base, e quando viene applicato

l'operatore `delete` a un puntatore del tipo della classe base che punta a un oggetto di una classe derivata, viene chiamato il distruttore della classe giusta. Ricordate che quando viene distrutto l'oggetto di una classe derivata viene distrutta anche la porzione della classe base presente in quell'oggetto, per cui il distruttore della classe base viene invocato automaticamente dopo il distruttore della classe derivata.

Buona abitudine 10.2

Se una classe contiene funzioni virtuali, includete anche un distruttore virtuale, anche se per tale classe non è necessario. Le classi derivate da essa potrebbero contenere distruttori che ne necessitano.

Errore tipico 10.2

I costruttori non possono essere virtuali: se dichiarate un costruttore come funzione virtuale commettete un errore di sintassi.

10.9 L'ereditarietà di interfaccia e di implementazione

Nel nostro prossimo esempio (Figura 10.2) riesaminiamo la gerarchia `Point`, `Circle` e `Cylinder` presentata nel capitolo precedente, ma ora al livello massimo della gerarchia poniamo la classe base astratta `Shape`. Essa contiene due funzioni virtuali pure, `printShapeName` e `print`, che la rendono classe base astratta, e due funzioni virtuali, `area` e `volume`, ognuna delle quali ha un'implementazione di default che restituisce il valore zero.

`Point` eredita queste implementazioni da `Shape` e non è necessario ridefinirle poiché sia l'area che il volume di un punto sono uguali a zero. `Circle` eredita la funzione `volume` da `Point`, ma fornisce la propria implementazione della funzione `area` mentre `Cylinder` fornisce le proprie implementazioni per entrambe le funzioni.

Osservate che, sebbene `Shape` sia una classe base astratta, contiene le implementazioni di alcune funzioni membro, ed esse possono essere ereditate. La classe `Shape` fornisce un'interfaccia composta da quattro funzioni virtuali, che saranno trasmesse a tutti i livelli della gerarchia. Inoltre `Shape` fornisce alcune implementazioni che saranno utilizzate dalle classi presenti nei primi livelli della gerarchia delle classi.

Ingegneria del software 10.8

Una classe può ereditare l'interfaccia o l'implementazione di una classe base. Nell'ereditarietà di implementazione le funzionalità vengono definite tendenzialmente ai livelli più alti della gerarchia: in questo modo le nuove classi derivate ereditano una o più funzioni membro definite nella classe base e ne usano le definizioni. Nell'ereditarietà di interfaccia le funzionalità vengono definite tendenzialmente nei livelli più bassi della gerarchia: in questo modo una classe base specifica una o più funzioni che devono essere chiamate da ogni oggetto della gerarchia (che hanno la stessa segnatura), ma ogni classe derivata ne fornisce la propria implementazione.


```
1  // Figura 10.2: shape.h
2  // Definizione della classe base astratta Shape
3  #ifndef SHAPE_H
4  #define SHAPE_H
5  #include <iostream.h>
6
7  class Shape {
8  public:
9     virtual double area() const { return 0.0; }
10    virtual double volume() const { return 0.0; }
11 // funzioni virtuali pure che subiranno ridefinizione nelle
12 // classi derivate
13    virtual void printShapeName() const = 0;
14    virtual void print() const = 0;
15 };
16
17 #endif
```

Figura 10.2 Definizione della classe base astratta Shape.

La classe base `Shape` è composta da quattro funzioni virtuali di tipo `public` e non contiene dati. Le funzioni `printShapeName` e `print` sono funzioni virtuali pure, per cui ogni classe derivata ne effettuerà la ridefinizione. Le funzioni `area` e `volume` restituiscono semplicemente `0.0` e saranno ridefinite nelle classi derivate, quando servirà avere valori diversi per l'area e il volume. Osservate che anche se `Shape` è una classe astratta, essa contiene alcune funzioni virtuali non pure (cioè `area` e `volume`). Le classi astratte, infatti, possono anche includere funzioni non virtuali e dati che saranno ereditati dalle classi derivate.

```
18 // Figura 10.2: point1.h
19 // Definizione della classe Point
20 #ifndef POINT1_H
21 #define POINT1_H
22 #include "shape.h"
23
24 class Point : public Shape {
25 public:
26    Point( int = 0, int = 0 );  // costruttore di default
27    void setPoint( int, int );
28    int getX() const { return x; }
29    int getY() const { return y; }
30    virtual void printShapeName() const { cout << "Point: "; }
31    virtual void print() const;
32 private:
33    int x, y;   // coordinate x,y di Point
34 };
35
36 #endif
```

Figura 10.2 Definizione della classe Point.

```
// Figura 10.2: point1.cpp
// Definizioni delle funzioni membro della classe Point
#include "point1.h"

Point::Point( int a, int b ) { setPoint( a, b ); }

void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

void Point::print() const
   { cout << '[' << x << ", " << y << ']'; }
```

Figura 10.2 Definizioni delle funzioni membro della classe `Point`.

La classe `Point` deriva da `Shape` con ereditarietà di tipo `public`. Un `Point` ha un'area e un volume nulli, per cui non c'è bisogno di ridefinire le funzioni `area` e `volume` della classe base: queste sono semplicemente ereditate così come sono state definite in `Shape`. Le funzioni `printShapeName` e `print` sono implementazioni di funzioni virtuali pure della classe base: se non ridefinissimo queste funzioni in `Point`, anch'essa sarebbe una classe astratta e non potremmo usarla per istanziare degli oggetti. Tra le altre funzioni membro troviamo una funzione *set* che assegna nuovi valori alle coordinate `x` e `y` e due funzioni *get* che restituiscono il valore di tali coordinate.

```
// Figura 10.2: circle1.h
// Definizione della classe Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
public:
   // costruttore di default
   Circle( double r = 0.0, int x = 0, int y = 0 );

   void setRadius( double );
   double getRadius() const;
   virtual double area() const;
   virtual void printShapeName() const { cout << "Circle: "; }
   virtual void print() const;
private:
   double radius;   // raggio
};

#endif
```

Figura 10.2 Definizione della classe `Circle`.


```
// Figura 10.2: circle1.cpp
// Definizioni delle funzioni membro della classe Circle
#include "circle1.h"

Circle::Circle( double r, int a, int b )
   : Point( a, b )  // chiama il costruttore della classe base
{ setRadius( r ); }

void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }

double Circle::getRadius() const { return radius; }

double Circle::area() const
   { return 3.14159 * radius * radius; }

void Circle::print() const
{
   Point::print();
   cout << "; Radius = " << radius;
}
```

Figura 10.2 Definizioni delle funzioni membro della classe `Circle`...

```
// Figura 10.2: cylindr1.h
// Definizione della classe Cylinder
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
public:
   // costruttore di default
   Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

   void setHeight( double );
   double getHeight();
   virtual double area() const;
   virtual double volume() const;
   virtual void printShapeName() const {cout << "Cylinder: ";}
   virtual void print() const;
private:
   double height;   // altezza
};

#endif
```

Figura 10.2 Definizione della classe `Cylinder`.

```
115  // Figura 10.2: cylindr1.cpp - Definizioni
116  // delle funzioni membro e friend della classe Cylinder
117  #include "cylindr1.h"
118
119  Cylinder::Cylinder( double h, double r, int x, int y )
120     : Circle( r, x, y ) // chiama il costruttore della classe base
121  { setHeight( h ); }
122
123  void Cylinder::setHeight( double h )
124     { height = h > 0 ? h : 0; }
125
126  double Cylinder::getHeight() { return height; }
127
128  double Cylinder::area() const
129  {
130     // area della superficie
131     return 2 * Circle::area() +
132            2 * 3.14159 * getRadius() * height;
133  }
134
135  double Cylinder::volume() const
136     { return Circle::area() * height; }
137
138  void Cylinder::print() const
139  {
140     Circle::print();
141     cout << "; Height = " << height;
142  }
```

Figura 10.2 Definizioni delle funzioni membro della classe `Cylinder`.

La classe `Circle` deriva da `Point` con ereditarietà di tipo `public`. Un `Circle` ha un volume nullo e, dunque, non c'è bisogno di ridefinire la funzione `volume` della classe base: essa è ereditata da `Point` la quale l'ha ereditata precedentemente da `Shape`. Un `Circle` però ha un area generalmente non nulla per cui qui occorre *ridefinire la funzione* `area`. Le funzioni `printShapeName` e `print` sono le implementazioni delle funzioni virtuali pure della classe `Shape`. Se non le ridefinissimo in questa classe, verrebbero ereditate le versioni definite in `Point`. Tra le altre funzioni membro troviamo una funzione *set* che assegna un nuovo valore al membro `radius` di `Circle` e una funzione *get* che ne restituisce il valore.

La classe `Cylinder` deriva da `Circle` con ereditarietà di tipo `public`. Un `Cylinder` ha area e volume diversi da quelli di `Circle` per cui in questa classe occorre ridefinire entrambe le funzioni `area` e `volume`. Le funzioni `printShapeName` e `print` sono le implementazioni delle funzioni virtuali pure della classe `Shape`. Se non le ridefinissimo in questa classe, verrebbero ereditate le versioni definite in `Circle`.

Tra le altre funzioni membro troviamo una funzione *set* che assegna un nuovo valore al membro `height` di `Cylinder` e una funzione *get* che ne restituisce il valore.


```
143  // Figura 10.2: fig10_02.cpp - Programma di esempio
144  // per la gerarchia shape, point, circle, cylinder
145  #include <iostream.h>
146  #include <iomanip.h>
147  #include "shape.h"
148  #include "point1.h"
149  #include "circle1.h"
150  #include "cylindr1.h"
151
152  void virtualViaPointer( const Shape * );
153  void virtualViaReference( const Shape & );
154
155  int main()
156  {
157     cout << setiosflags( ios::fixed | ios::showpoint )
158          << setprecision( 2 );
159
160     Point point( 7, 11 );                  // crea un Point
161     Circle circle( 3.5, 22, 8 );           // crea un Circle
162     Cylinder cylinder( 10, 3.3, 10, 10 );  // crea un Cylinder
163
164     point.printShapeName();    // binding statico
165     point.print();             // binding statico
166     cout << '\n';
167
168     circle.printShapeName();   // binding statico
169     circle.print();            // binding statico
170     cout << '\n';
171
172     cylinder.printShapeName(); // binding statico
173     cylinder.print();          // binding statico
174     cout << "\n\n";
175
176     Shape *arrayOfShapes[ 3 ]; //array di puntatori di classe base
177
178  // punta arrayOfShapes[0] all'oggetto della classe derivata Point
179     arrayOfShapes[ 0 ] = &point;
180
181  // punta arrayOfShapes[1] all'oggetto della classe der. Circle
182     arrayOfShapes[ 1 ] = &circle;
183
184  // punta arrayOfShapes[2] all'oggetto della classe der. Cylinder
185     arrayOfShapes[ 2 ] = &cylinder;
186
187  // Ciclo che attraversa arrayOfShapes e chiama virtualViaPointer
188  // per visualizzare il nome, gli attributi, l'area e il volume
189  // di ogni oggetto utilizzando il binding dinamico.
```

Figura 10.2 Programma di esempio della gerarchia `Point`, `Circle`, `Cylinder` (continua).

```
190     cout << "Virtual function calls made off "
191          << "base-class pointers\n";
192
193     for ( int i = 0; i < 3; i++ )
194        virtualViaPointer( arrayOfShapes[ i ] );
195
196  // Ciclo che attraversa arrayOfShapes e chiama virtualViaPointer
197  // per visualizzare il nome, gli attributi, l'area e il volume
198  // di ogni oggetto utilizzando il binding dinamico.
199     cout << "Virtual function calls made off "
200          << "base-class references\n";
201
202     for ( int j = 0; j < 3; j++ )
203        virtualViaReference( *arrayOfShapes[ j ] );
204
205     return 0;
206  }
207
208  // Chiamate a funzioni virtuali effettuate tramite un puntatore
209  // di classe base utilizzando il binding dinamico.
210  void virtualViaPointer( const Shape *baseClassPtr )
211  {
212     baseClassPtr->printShapeName();
213     baseClassPtr->print();
214     cout << "\nArea = " << baseClassPtr->area()
215          << "\nVolume = " << baseClassPtr->volume() << "\n\n";
216  }
217
218  // Chiamate a funzioni virtuali effettuate tramite un riferimento
219  // di classe base utilizzando il binding dinamico.
220  void virtualViaReference( const Shape &baseClassRef )
221  {
222     baseClassRef.printShapeName();
223     baseClassRef.print();
224     cout << "\nArea = " << baseClassRef.area()
225          << "\nVolume = " << baseClassRef.volume() << "\n\n";
226  }
```

```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00

Virtual function calls made off base-class pointers
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

Figura 10.2 Programma di esempio della gerarchia `Point`, `Circle`, `Cylinder` (continua).


```
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

Figura 10.2 Programma di esempio della gerarchia Point, Circle, Cylinder.

Il programma di esempio (fig10_02.cpp) istanzia un oggetto `Point` di nome `point`, un oggetto `Circle` di nome `circle` e un oggetto `Cylinder` di nome `cylinder`. Le funzioni `printShapeName` e `print` sono quindi invocate su ognuno di essi, per visualizzarne il nome e mostrare che sono stati inizializzati correttamente. Ogni chiamata a `printShapeName` e `print` nelle linee 164-173 utilizza il binding statico, perché il compilatore sa già durante la compilazione il tipo di ciascun oggetto sul quale sono chiamate `printShapeName` e `print`.

Successivamente viene dichiarato l'array `arrayOfShapes`, ciascun elemento del quale è di tipo `Shape *`. Questo array di puntatori di classe base serve a puntare agli oggetti delle classi derivate; infatti, non è possibile definire direttamente oggetti di tipo `Shape` poiché essa è una classe astratta. L'indirizzo di `point` è assegnato a `arrayOfShapes[ 0 ]` (linea 179), quello di `circle` a `arrayOfShapes[ 1 ]` (linea 182) e, infine, quello di `cylinder` a `arrayOfShapes[ 2 ]` (linea 185).

Subito dopo, un ciclo `for` (linea 193) scandisce l'array `arrayOfShapes` e invoca la funzione `virtualViaPointer` (linea 194) per ogni elemento dell'array:

```
virtualViaPointer( arrayOfShapes[ i ] );
```

La funzione `virtualViaPointer` riceve nel parametro `baseClassPtr` (di tipo `const Shape *`) l'indirizzo memorizzato in un elemento di `arrayOfShapes` e, per ciascuno di essi, sono effettuate le seguenti chiamate di funzioni virtuali

```
baseClassPtr->printShapeName()
baseClassPtr->print()
baseClassPtr->area()
baseClassPtr->volume()
```

Ogni riga invoca una funzione virtuale sull'oggetto a cui punta `baseClassPtr` durante l'esecuzione, il cui tipo non può essere determinato durante la compilazione. L'output illustra che sono invocate le giuste funzioni per ogni classe. Sono visualizzate per prime la stringa `"Point: "` e le coordinate dell'oggetto `point`; area e volume sono entrambi pari a `0.00`. In seguito, sono visualizzati la stringa `"Circle: "`, le coordinate del centro dell'oggetto `circle` e il valore del raggio (`radius`); viene poi calcolata l'area mentre il valore restituito per il volume è `0.00`. Poi sono visualizzati la stringa `"Cylinder: "`, le coordinate del centro della base dell'oggetto `cylinder`, il valore del raggio `radius` e l'altezza `height`; infine sono calcolati i valori di area e volume. Tutte le chiamate alle funzioni virtuali `printShapeName`, `print`, `area` e `volume` sono risolte durante l'esecuzione attraverso il binding dinamico.

Infine un ciclo `for` (linea 202) attraversa `arrayOfShapes` e invoca la funzione `virtualViaReference` (linea 203).

```
virtualViaReference( *arrayOfShapes[ i ] );
```

per ogni elemento dell'array. La funzione `virtualViaReference` riceve nel suo parametro `baseClassRef` (di tipo `const Shape &`) un riferimento formato dereferenziando l'indirizzo contenuto in un elemento dell'array. In ogni chiamata a `virtualViaReference`, sono effettuate le seguenti chiamate a funzioni virtuali

```
baseClassRef.printShapeName()
baseClassRef.print()
baseClassRef.area()
baseClassRef.volume()
```

Ogni linea invoca queste funzioni per l'oggetto riferito da `baseClassRef`; l'output generato quando si utilizzano riferimenti di classe base è identico a quello generato quando si utilizzano puntatori di classe base.

10.10 L'implementazione di polimorfismo, funzioni virtuali e binding dinamico

Il C++ consente di introdurre il polimorfismo nei programmi in modo semplice e diretto. È certamente possibile introdurlo anche in altri linguaggi non orientati agli oggetti, come il C, ma al costo di manipolazioni su puntatori complesse e potenzialmente rischiose. In questa sezione parliamo del modo in cui il C++ implementa internamente il polimorfismo, le funzioni virtuali e il binding dinamico. Speriamo che acquisiate una conoscenza solida dell'argomento e del funzionamento reale di queste caratteristiche del linguaggio. Sbirciando dietro le quinte, inoltre, diverrete consapevoli del consumo di risorse che comporta il polimorfismo, in termini di tempo e di memoria, per cui saprete discernere con maggior cognizione di causa quando utilizzarlo e quando evitarlo.

Per cominciare spiegheremo le strutture dati che il compilatore crea durante la compilazione per supportare il polimorfismo a tempo di esecuzione, dopo di che mostreremo come un programma in esecuzione si serve di queste strutture dati per eseguire le funzioni virtuali ed effettuare il binding dinamico.

Quando il C++ compila una classe che contiene una o più funzioni virtuali, crea una tabella di funzioni virtuali per quella determinata classe detta *vtable*. Essa viene utilizzata durante l'esecuzione del programma per selezionare le implementazioni appropriate ogni volta che deve essere chiamata una funzione virtuale di quella classe. La Figura 10.3 illustra le vtable per le classi `Shape`, `Point`, `Circle` e `Cylinder`.

Nella vtable di `Shape`, il primo puntatore punta all'implementazione della funzione `area` di tale classe, una funzione che restituisce un valore dell'area pari a `0.0` e il secondo puntatore punta alla funzione `volume`, che restituisce anch'essa `0.0`. Le funzioni `printShapeName` e `print` sono funzioni virtuali pure, cioè non hanno un'implementazione, per cui i puntatori relativi a esse sono impostati entrambi a `0`: ogni classe che ha uno o più puntatori uguali nulli nella sua vtable è una classe astratta. Le classi che non contengono puntatori nulli nella vtable (come `Point`, `Circle` e `Cylinder`) sono invece classi concrete.

La classe `Point` eredita le funzioni `area` e `volume` dalla classe `Shape`, per cui il compilatore imposta semplicemente questi due puntatori nella vtable di `Point` come copie dei puntatori alle funzioni `area` e `volume` presenti nella vtable di `Shape`. La classe `Point` ridefinisce la funzione `printShapeName` per poter visualizzare `"Point: "`, per cui il puntatore corrispondente punta alla funzione `printShapeName` della classe `Point`. Inoltre, a questo livello viene ridefinito anche `print`, per cui il puntatore corrispondente punta alla funzione della classe `Point` che visualizza `[ x, y ]`.

Il puntatore alla funzione `area` nella vtable di `Circle` punta alla funzione `area` di `Circle` che restituisce πr^2. Il puntatore alla funzione `volume` è una semplice copia del corrispondente presente nella classe `Point` che a sua volta era stato già copiato da `Shape`. Il puntatore a `printShapeName` punta alla versione di `Circle` della funzione, che visualizza `"Circle: "`. Il puntatore a `print` punta alla funzione `print` di `Circle`, che visualizza `[ x, y ]` *r*.

Il puntatore alla funzione `area` nella vtable di `Cylinder` punta alla funzione `area` di `Cylinder`, che calcola l'area della superficie con la formula $2\pi r^2 + 2\pi rh$. Il puntatore a `volume` di `Cylinder` punta alla funzione `volume` che restituisce $\pi r^2 h$. Il puntatore a `printShapeName` di `Cylinder` punta alla funzione che visualizza `"Cylinder: "` mentre quello a `print` di `Cylinder` punta alla propria funzione, che visualizza `[ x, y ]` *r h*.

Il polimorfismo viene realizzato con una struttura dati complessa che coinvolge tre livelli di puntatori. Finora abbiamo esaminato solo il livello relativo ai puntatori a funzione della vtable. Tali puntatori puntano alle funzioni reali che devono essere eseguite quando viene invocata una funzione virtuale.

Passiamo ora al secondo livello di puntatori: quando si istanzia un oggetto di una classe che contiene funzioni virtuali, il compilatore appone all'oggetto un puntatore alla vtable della classe. Normalmente il puntatore viene apposto all'oggetto, ma non è necessaria un'implementazione di questo tipo.

Il terzo livello di puntatori consiste semplicemente nell'handle dell'oggetto che riceve la chiamata alla funzione virtuale (questo handle può anche essere un riferimento).

Adesso vediamo come viene eseguita una tipica chiamata a una funzione virtuale. Consideriamo la chiamata

```
baseClassPtr->printShapeName()
```

in `virtualViaPointer`. Supponiamo, per il resto della discussione, che `baseClassPtr` contenga l'indirizzo che si trova in `arrayOfShapes[ 1 ]`, cioè l'indirizzo dell'oggetto `circle`. Quando il compilatore incontra questa istruzione, determina che la chiamata è stata effettuata *tramite un puntatore di classe base e che `printShapeName` è una funzione virtuale.*

A questo punto il compilatore determina che `printShapeName` è la terza voce in ogni vtable e per localizzarla il compilatore sa quindi che dovrà saltare le prime due. Perciò, nel codice oggetto che effettuerà questa chiamata di funzione virtuale, essa viene compilata come un *offset* o *scostamento* di 8 byte (supponendo ciascun puntatore occupi esattamente 4 byte come accade sulle moderne macchine a 32 bit).

Quindi il compilatore genera il codice che: (NB: i numeri nell'elenco corrispondono ai numeri nei cerchietti in Figura 10.3):

1. Seleziona l'*i*-esima voce di `arrayOfShapes` (in questo caso l'indirizzo dell'oggetto `circle`) e la passa alla funzione `virtualViaPointer`. Ciò fa sì che `baseClassPtr` punti a `circle`.
2. Dereferenzia tale puntatore per ottenere l'oggetto `circle` che, come ricorderete, inizia con un puntatore alla vtable di `Circle`.
3. Dereferenzia il puntatore nella vtable di `circle` per ottenere la vtable di `Circle`.
4. *Salta l'offset di 8 byte per ricavare il puntatore alla funzione `printShapeName`.*
5. Dereferenzia il puntatore alla funzione `printShapeName` per formare il nome della funzione reale da eseguire e utilizza l'operatore di chiamata di funzione `()` per eseguire la giusta versione di `printShapeName`, che visualizza la stringa `"Circle: "`.

Le strutture dati in Figura 10.3 possono sembrare complesse, ma la totalità di questa complessità è gestita autonomamente dal compilatore ed è nascosta al programmatore, rendendo l'implementazione del polimorfismo in C++ semplice e diretta.

Le operazioni di manipolazione di puntatori e di accesso alla memoria necessarie per ogni chiamata a una funzione virtuale naturalmente costituiscono un costo in termini di tempo durante l'esecuzione: inoltre le vtable e i puntatori alle vtable aggiunti agli oggetti richiedono memoria aggiuntiva.

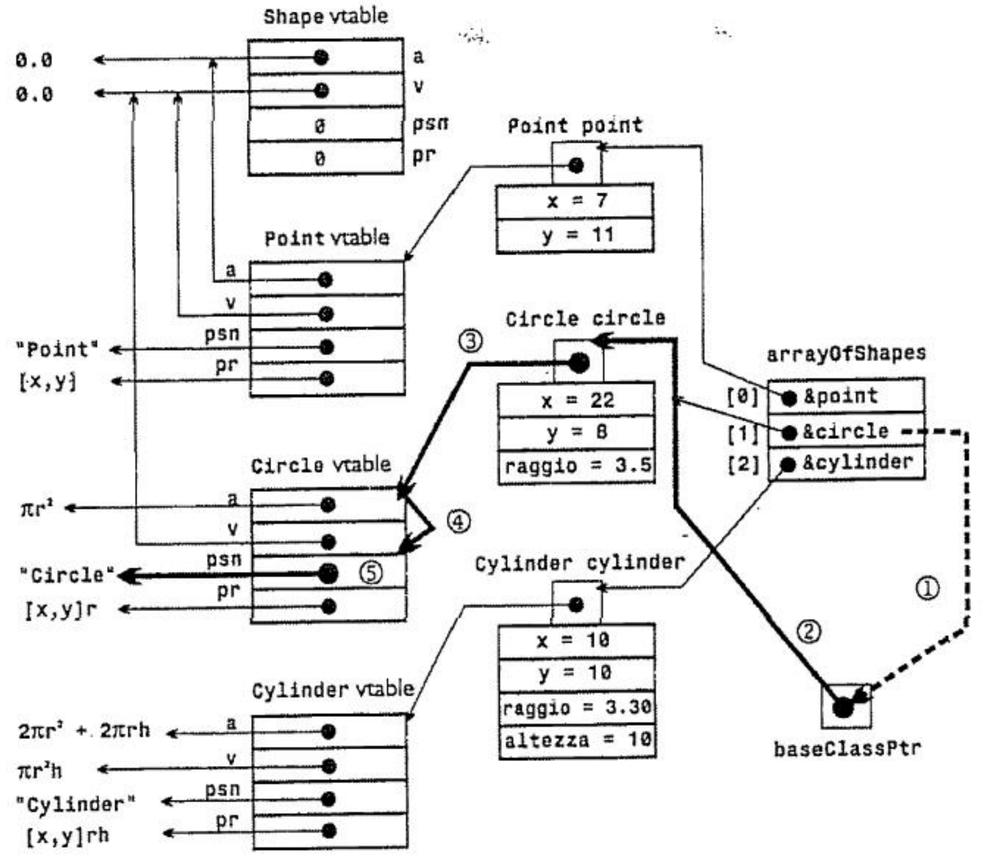
A questo punto speriamo che ne sappiate abbastanza sulle funzioni virtuali per determinare caso per caso se conviene utilizzarle nelle vostre applicazioni.

Obiettivo efficienza 10.1

Il polimorfismo implementato tramite funzioni virtuali e binding dinamico è efficiente. L'impatto che ha sul tempo di esecuzione può considerarsi trascurabile.

Obiettivo efficienza 10.2

Funzioni virtuali e binding dinamico consentono di sostituire la logica dello `switch` con quella del polimorfismo. I moderni compilatori creano normalmente codice ottimizzato almeno altrettanto efficiente di quello generato con la logica dello `switch`. In un modo o nell'altro, la perdita di efficienza associata al polimorfismo è accettabile per la maggior parte delle applicazioni. Tuttavia in alcune situazioni, fra cui le applicazioni in real-time che hanno stringenti necessità di efficienza, questa perdita può rivelarsi troppo elevata.



`a` = funzione `area`
`v` = funzione `volume`
`psn` = funzione `printShapeName`
`pr` = funzione `print`
`0` significa funzione virtuale pura
`r` = `raggio` `h` = `altezza`

Il diagramma della chiamata alla funzione virtuale `baseClassPtr->printShapeName()` è illustrato qui sopra dalle frecce in grassetto

① passa `&circle` a `baseClassPtr`
② per ottenere l'oggetto `Circle`
③ per ottenere la vtable di `Circle`
④ per ricavare il puntatore a `printShapeName` nella vtable
⑤ esegue `printShapeName` per `Circle`

Figura 10.3 Flusso di controllo di una chiamata a una funzione virtuale.

Esercizi di autovalutazione

10.1 Completate le seguenti affermazioni:
a) L'ereditarietà e il polimorfismo sostituiscono la logica di _____.
b) Una funzione virtuale pura si specifica scrivendo _____ al termine del suo prototipo nella definizione di classe.

c) Se una classe contiene una o più funzioni virtuali pure, è una _____.
d) Se una chiamata di funzione viene risolta durante la compilazione si ha il binding _____.
e) Se una chiamata di funzione viene risolta durante la compilazione si ha il binding _____.

Risposte agli esercizi di autovalutazione

10.1 a) `switch`. b) = 0. c) classe base astratta. d) statico. e) dinamico.

Esercizi

10.2 Che cosa sono le funzioni virtuali? Descrivete una circostanza in cui è appropriato utilizzarle.

10.3 Ricordando che i costruttori non possono essere virtuali, date uno schema di come potreste realizzare un effetto simile.

10.4 Spiegate in che modo il polimorfismo consente di programmare "in generale" anziché "nello specifico". Discutete i vantaggi principali di questo tipo di programmazione.

10.5 Discutete le problematiche associate alla logica dello `switch`. Spiegate in che modo il polimorfismo ne costituisce una valida alternativa.

10.6 Indicate le differenze tra binding statico e dinamico. Spiegate l'utilizzo delle funzioni virtuali e delle vtable nel binding dinamico.

10.7 Indicate le differenze tra ereditarietà di interfaccia e di implementazione. In che cosa differiscono le gerarchie progettate per il primo tipo di ereditarietà da quelle progettate per il secondo?

10.8 Indicate le differenze tra funzioni virtuali e funzioni virtuali pure.

10.9 (Vero/Falso) Tutte le funzioni virtuali di una classe base astratta devono essere dichiarate come funzioni virtuali pure.

10.10 Suggerite uno o più livelli di classi base astratte per la gerarchia di `Shape` proposta in questo capitolo (il primo livello è `Shape` e il secondo comprende `TwoDimensionalShape` e `ThreeDimensionalShape`).

10.11 In che senso il polimorfismo promuove l'estensibilità del software?

10.12 Vi è stato chiesto di scrivere un simulatore di volo che prevede un output grafico elaborato. Spiegate in che modo il polimorfismo può rivelarsi particolarmente adatto a questo genere di problemi.

10.13 Scrivete un semplice pacchetto grafico utilizzando la gerarchia della classe `Shape` del Capitolo 9. Limitatevi alle forme geometriche bidimensionali come quadrati, rettangoli, triangoli e cerchi. Interagite con l'utente consentendogli di specificare la posizione, la dimensione, la forma e il carattere di riempimento per ogni forma geometrica. L'utente può specificare di tracciare molte figure geometriche dello stesso tipo; alla creazione di ogni figura ponete il suo indirizzo in un array con elementi di tipo `Shape *`. Ogni classe contiene la propria versione della funzione membro `draw` per il tracciamento della figura. Scrivete uno screen manager polimorfo che attraversa l'array (meglio se con un iteratore) inviando messaggi `draw` a ciascun oggetto dell'array, per formare un'immagine sullo schermo. Tracciate nuovamente l'immagine dello schermo ogni volta che l'utente specifica una nuova figura geometrica.

10.14 Modificate il libro paga elettronico in Figura 10.1, aggiungendo alla classe `Employee` i dati membro `private` `birthDate` (data di nascita, un oggetto `Date`) e `departmentCode` (codice del dipartimento, un `int`). Supponete che questo libro paga sia eseguito una volta al mese. Nel calcolo (polimorfo) dei guadagni di ciascun `Employee`, aggiungete un bonus di `$100.00` se nel mese corrente cade il compleanno dell'`Employee`.

10.15 Nell'Esercizio 9.14 avete sviluppato e definito le classi della gerarchia `Shape`. Modificate la gerarchia in modo che la classe `Shape` sia una classe base astratta che contiene l'interfaccia della gerarchia e derivate da essa le classi astratte `TwoDimensionalShape` e `ThreeDimensionalShape`. Utilizzate la funzione virtuale `print` per visualizzare il tipo e le dimensioni di ogni classe. Prevedete inoltre le funzioni virtuali `area` e `volume`, in modo che questi calcoli possano essere effettuati per gli oggetti delle classi concrete della gerarchia e, infine, scrivete un programma di esempio che utilizzi la gerarchia `Shape`.

CAPITOLO 11

Gli stream di input/output del C++

Obiettivi

- Imparare a utilizzare gli stream di input/output del C++
- Imparare a formattare l'input e l'output
- Comprendere la gerarchia delle classi stream per l'I/O
- Imparare a effettuare l'input/output dei tipi definiti dall'utente
- Imparare a creare manipolatori di stream definiti dall'utente
- Imparare a determinare se un'operazione di I/O è riuscita
- Imparare a collegare uno stream di output a uno stream di input

11.1 Introduzione

Le librerie standard del C++ forniscono un insieme cospicuo di funzionalità per le operazioni di input/output. In questo capitolo approfondiremo questo discorso esaminandone un sottoinsieme sufficiente per effettuare le operazioni di I/O più comuni e passando brevemente in rassegna le altre. Alcune delle funzionalità che presentiamo sono state già utilizzate nei capitoli precedenti, ma ne riprendiamo la trattazione per rendere il discorso sull'I/O più completo e organico.

Molte delle funzionalità di I/O sono orientate agli oggetti ed è interessante comprendere come sono implementate: questo stile di I/O si serve di altre caratteristiche del C++ come i riferimenti o l'overloading delle funzioni e degli operatori.

L'I/O del C++ è *type safe*, cioè salvaguarda l'integrità dei tipi di dato: ogni operazione di I/O è effettuata automaticamente in funzione del tipo di dato che tratta. Se si definisce correttamente una funzione di I/O per un particolare tipo di dato, essa sarà chiamata quando occorre gestire quel tipo di dato mentre se non viene trovata alcuna funzione definita per trattarlo, il compilatore segnalerà un errore. Ciò significa che non è possibile l'output di dati scorretti, cosa possibile invece in C dove, a volte, si possono verificare errori sottili e alquanto bizzarri.

Come abbiamo detto, in C++ è possibile specificare l'I/O dei tipi definiti dall'utente così come dei tipi standard e questo costituisce un altro esempio dell'estensibilità del linguaggio.

Buona abitudine 11.1

Nei programmi scritti in C++ utilizzate esclusivamente le funzioni per l'I/O del C++, anche se sono ancora disponibili le funzioni del C.

Ingegneria del software 11.1

L'I/O nello stile C++ salvaguarda l'integrità dei tipi di dato.

Ingegneria del software 11.2

Il C++ prevede un trattamento simile per l'I/O dei tipi predefiniti e dei tipi definiti dall'utente. Ciò facilita lo sviluppo e il riutilizzo del software.

11.2 Gli stream

L'I/O in C++ avviene per mezzo di *stream* (o *flussi*), costituiti semplicemente da una sequenza di byte. Nelle operazioni di input, i byte fluiscono da un dispositivo (per es., la tastiera, l'hard disk o una connessione di rete) alla memoria principale, mentre in quelle di output i byte fluiscono nella direzione opposta.

Sono le applicazioni che associano un significato ai byte; essi, infatti, possono rappresentare caratteri ASCII, dati interni in formato grezzo, immagini grafiche, testo parlato, video digitale o qualsiasi altro tipo di informazione.

Il compito dei meccanismi di I/O di sistema è trasferire i byte dai dispositivi alla memoria e viceversa in maniera coerente e affidabile. Questi trasferimenti spesso coinvolgono eventi meccanici, come la rotazione di un disco o di un nastro, o la digitazione di lettera alla tastiera ed il tempo da loro richiesto è tipicamente enorme se confrontato con il tempo in cui il processore elabora internamente i dati. Perciò le operazioni di I/O richiedono una pianificazione e un'ottimizzazione intelligenti per raggiungere un buon livello di efficienza. Le problematiche di questo genere sono affrontate approfonditamente nei libri di testo sui sistemi operativi.

Il C++ fornisce funzionalità di I/O sia di basso che di alto livello. Le prime si riferiscono all'I/O non formattato e normalmente specificano semplicemente che un certo numero di byte deve essere trasferito dalla memoria a un dispositivo o viceversa concentrandosi sul singolo byte. Anche se l'I/O a basso livello è ad alta velocità e consente di trattare efficientemente grandi volumi di dati, non è particolarmente significativo per gli esseri umani.

Le persone, infatti, preferiscono una visione ad alto livello dell'I/O (cioè l'I/O formattato) in cui i byte sono raggruppati in unità significative come gli interi, i numeri a virgola mobile, i caratteri, le stringhe e i tipi definiti dall'utente. Le funzionalità di I/O ad alto livello sono soddisfacenti per la maggior parte delle applicazioni, tranne che per quelle che richiedono di trattare efficientemente grandi volumi di dati.

Obiettivo efficienza 11.1

Utilizzate l'I/O non formattato per ottimizzare l'elaborazione di operazioni che trattano grandi volumi di dati.

11.2.1 I file di intestazione della libreria iostream

La libreria `iostream` fornisce centinaia di funzionalità per l'I/O. Ci sono diversi file di intestazione che contengono porzioni dell'interfaccia della libreria.

La maggior parte dei programmi in C++ include il file di intestazione `<iostream.h>` che contiene le informazioni di base per tutte le operazioni di I/O su stream. Il file `<iostream.h>` contiene gli oggetti `cin`, `cout`, `cerr` e `clog` che corrispondono agli stream standard di input, di output, di errore non bufferizzato e di errore bufferizzato. Sono presenti funzionalità di I/O formattato e non formattato.

Il file `<iomanip.h>` contiene informazioni utili per effettuare l'I/O formattato con i cosiddetti *manipolatori di stream parametrizzati*, mentre il file `<fstream.h>` contiene informazioni importanti per l'elaborazione dei file controllata dall'utente. Ogni implementazione del C++ contiene generalmente altre librerie correlate all'I/O che forniscono funzionalità specifiche dei diversi sistemi, come il controllo di periferiche speciali per l'I/O di audio e video.

11.2.2 Le classi e gli oggetti che effettuano l'input/output su stream

La libreria `iostream` contiene diverse classi per il trattamento di una vasta gamma di operazioni di I/O. Le classi `istream` e `ostream` supportano rispettivamente le operazioni di input ed output su stream mentre la classe `iostream` supporta entrambi i tipi di operazioni. Le classi `istream` e `ostream` derivano ognuna con ereditarietà singola dalla classe base `ios` e la classe `iostream` deriva con ereditarietà multipla da quest'ultime. Queste relazioni sono riepilogate nella Figura 11.1.

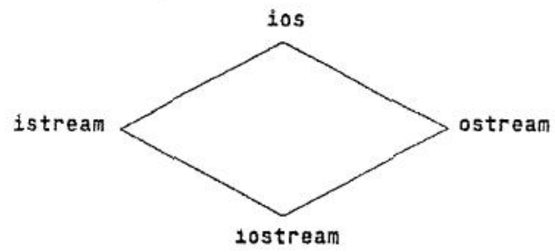


Figura 11.1 Gerarchia parziale delle classi che effettuano l'I/O su stream.

L'overloading degli operatori consente di utilizzare una notazione conveniente per le operazioni di input/output. L'overloading dell'operatore di shifting a sinistra (`<<`) invia l'output sullo stream e prende il nome di *operatore di inserimento nello stream*. L'overloading dell'operatore di shifting a destra (`>>`) estrae l'input dallo stream e prende il nome di *operatore di estrazione dallo stream*. Questi operatori si utilizzano con gli oggetti degli stream standard `cin`, `cout`, `cerr` e `clog`, e comunemente con gli oggetti stream definiti dall'utente.

`cin` è un oggetto della classe `istream` ed è *associato* o *connesso* con il dispositivo di input standard, normalmente la tastiera. Usato nel modo seguente, l'operatore di estrazione dallo stream causa l'input di un valore intero per la variabile intera `grade` da `cin` alla memoria:

```
cin >> grade;
```

Notate che, grazie all'overloading, l'operazione di estrazione dallo stream è abbastanza intelligente da *conoscere* di che tipo di dato si tratta. Supponendo di aver dichiarato correttamente `grade`, non abbiamo bisogno di fornire informazioni aggiuntive sul tipo di dato che *deve essere estratto dallo stream di input* (come accade, invece, nell'I/O in stile C).

`cout` è un oggetto della classe `ostream` ed è associato con il dispositivo di output standard, normalmente lo schermo. L'operatore di inserimento nello stream, usato nel modo seguente, causa l'output del valore intero di `grade` dalla memoria al dispositivo di output standard:

```
cout << grade;
```

Nuovamente, grazie all'overloading, l'operazione di inserimento nello stream è "abbastanza intelligente" da *conoscere* di che tipo di dato si tratta. Supponendo di aver dichiarato correttamente `grade`, non abbiamo bisogno di fornire informazioni aggiuntive sul tipo di dato che deve essere inviato sullo stream di output.

`cerr` è un oggetto della classe `ostream` ed è associato al dispositivo di errore standard. Gli output su `cerr` non sono bufferizzati: ciò significa che *ogni operazione di inserimento* nello stream effettuata su `cerr` causa un output immediato. Questo comportamento serve a notificare istantaneamente un errore all'utente.

`clog` è un oggetto della classe `ostream` ed è anch'esso associato al dispositivo di errore standard ma, a differenza di `cerr`, gli output su `clog` sono bufferizzati: ciò significa che ogni inserimento in `clog` non causerà un output immediato, a meno che il buffer non si riempia o venga esplicitamente *svuotato* (*con un operazione detta di flush*).

L'elaborazione dei file in C++ si serve delle classi `ifstream` per l'input da file, `ofstream` per l'output su file e `fstream` per l'input/output da/su file. La classe `ifstream` deriva da `istream`, la classe `ofstream` da `ostream` e la classe `fstream` da `iostream`. Le varie relazioni di ereditarietà delle classi relative all'I/O sono riepilogate nella Figura 11.2.

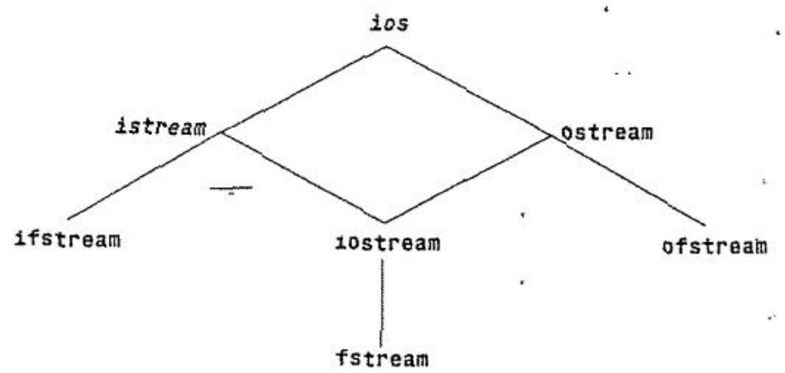


Figura 11.2 *Gerarchia parziale delle classi per l'I/O su stream, comprendente le classi* principali per l'elaborazione dei file.

Per la maggior parte delle installazioni sono presenti molte più classi nella gerarchia completa per l'I/O su stream, ma le classi che mostriamo qui forniscono le funzionalità richieste dalla maggioranza dei programmi.

11.3 L'output su stream

La classe `ostream` fornisce le funzionalità per effettuare l'output formattato e non formattato. Tra le funzionalità di output troviamo:

- l'output dei tipi standard tramite l'operatore di inserimento nello stream;
- l'output di caratteri tramite la funzione membro `put`;
- l'output non formattato tramite la funzione membro `write` (Sezione 11.5);
- l'output di interi in formato decimale, ottale ed esadecimale (Sezione 11.6.1);
- l'*output di valori a virgola mobile con diverse precisioni* (Sezione 11.6.2), *con il punto* decimale forzato (Sezione 11.7.2), nella notazione scientifica e nella notazione fissa (Sezione 11.7.6);
- l'output dei dati giustificati in campi di ampiezza determinata (Sezione 11.7.3);
- l'output dei dati in campi completati con caratteri di riempimento specificati (Sezione 11.7.4);
- l'output di lettere maiuscole nella notazione scientifica ed esadecimale (Sezione 11.7.7).

11.3.1 L'operatore di inserimento nello stream

L'output su stream si può effettuare tramite l'operatore di inserimento nello stream, ovvero l'overloading dell'operatore `<<`. Esso effettua l'output dei tipi predefiniti, delle stringhe e dei puntatori e nella Sezione 11.9 vedremo come effettuarne l'overloading per l'output dei tipi di dato definiti dall'utente.

Il programma nella Figura 11.3 mostra l'output di una stringa tramite un'istruzione di inserimento nello stream. Si possono utilizzare più istruzioni di inserimento, come vedete nella Figura 11.4. Quando viene eseguito il secondo programma, produce lo stesso output del primo.

```
1  // Fig. 11.3: fig11_03.cpp
2  // Output di una stringautilizzando l'inserimento nello stream.
3  #include <iostream.h>
4
5  int main()
6  {
7     cout << "Welcome to C++!\n";
8
9     return 0;
10 }
```

```
Welcome to C++!
```

Figura 11.3 Output di una stringa tramite l'inserimento nello stream.

```
1   // Fig. 11.4: fig11_04.cpp
2   // Output di una stringa utilizzando 2 inserimenti nello stream.
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "Welcome to ";
8      cout << "C++!\n";
9
10     return 0;
11  }
```

```
Welcome to C++!
```

Figura 11.4 Output di una stringa tramite due inserzioni nello stream.

L'effetto della sequenza `\n` (nuova linea) si ottiene anche tramite `endl` (in inglese *end line*, fine della linea) che è un *manipolatore di stream* (presentato nella Figura 11.5). Esso emette un carattere di nuova linea e svuota il buffer, cioè causa l'output del contenuto del buffer anche se questo non è ancora pieno. Lo svuotamento del buffer di output si può effettuare anche nel modo seguente:

```
cout << flush;
```

I manipolatori di stream sono discussi in dettaglio nella Sezione 11.6.

Le espressioni possono essere inviate in output nel modo mostrato nella Figura 11.6.

Buona abitudine 11.2

Se effettuate l'output di espressioni, ricordatevi di porle tra parentesi per evitare problemi di precedenza tra gli operatori presenti nell'espressione e l'operatore <<.

```
1   // Fig. 11.5: fig11_05.cpp
2   // Utilizzo del manipolatore di stream endl.
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "Welcome to ";
8      cout << "C++!";
9      cout << endl;    // manipolatore di stream endl
10
11     return 0;
12  }
```

```
Welcome to C++!
```

Figura 11.5 Il manipolatore di stream endl.


```
1   // Fig. 11.6: fig11_06.cpp
2   // Output dei valori di un'espressione.
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "47 plus 53 is ";
8
9   // le parentesi non sono necessarie, ma sono usate per chiarezza
10     cout << ( 47 + 53 );      // espressione
11     cout << endl;
12
13     return 0;
14  }
```

```
47 plus 53 is 100
```

Figura 11.6 Output di espressioni.

11.3.2 Utilizzo degli operatori di inserimento/estrazione in cascata

Le operazioni di << e >> possono essere scritte in cascata, come mostra la Fig. 11.7.

```
1   // Fig. 11.7: fig11_07.cpp
2   // Utilizzo dell'overloading dell'operatore << in cascata.
3   #include <iostream.h>
4
5   int main()
6   {
7      cout << "47 plus 53 is " << ( 47 + 53 ) << endl;
8
9      return 0;
10  }
```

```
47 plus 53 is 100
```

Figura 11.7 Uso dell'operatore << in cascata.

Le operazioni di inserimento nella Figura 11.7 sono eseguite come se fossero state scritte nella forma:

```
( ( ( cout << "47 plus 53 is " ) << ( 47 + 53 ) ) << endl );
```

cioè << associa da sinistra a destra. Le operazioni in cascata sono possibili perché l'operatore << restituisce un riferimento al suo operando sinistro, `cout`. Perciò l'espressione di sinistra tra parentesi

```
( cout << "47 plus 53 is " )
```

invia in output la stringa di caratteri specificata e restituisce un riferimento a `cout`. Questo significa che l'espressione centrale tra parentesi viene valutata come

```
( cout << ( 47 + 53 ) )
```

che invia in output il valore intero `100` e restituisce un riferimento a `cout`. L'espressione di destra tra parentesi viene valutata poi come

```
cout << endl
```

che invia in output un carattere di nuova linea, svuota il buffer di `cout` e restituisce un riferimento a `cout`, che rimane inutilizzato.

11.3.3 L'output delle variabili di tipo char*

Nell'I/O in stile C è necessario che il programmatore fornisca informazioni sui tipi dei dati che, invece, il C++ determina automaticamente. A volte però, in un certo senso, il C++ fa più del dovuto. Per esempio, sappiamo che una stringa di caratteri è in realtà un puntatore di tipo `char *`. Supponiamo di voler visualizzare il valore di tale puntatore, *ovvero dell'indirizzo di memoria del primo carattere della stringa; per far questo ci scontriamo* con l'overloading dell'operatore `<<` che prevede la visualizzazione del tipo `char *` come stringa terminata da un carattere nullo. La soluzione consiste nell'effettuare un cast del puntatore in `void *` (questa operazione è valida per qualsiasi puntatore di cui si voglia visualizzare il valore). Il programma nella Figura 11.8 mostra la visualizzazione di una variabile `char *` in formato stringa e in formato indirizzo. Osservate che l'indirizzo viene visualizzato come un numero esadecimale (in base 16). In C++ i numeri esadecimali iniziano con il prefisso `0x` o `0X`. Ritorneremo su questi concetti nelle Sezioni 11.6.1, 11.7.4, 11.7.5 e 11.7.7.

```
1  // Fig. 11.8: fig11_08.cpp - Visualizzazione dell'indirizzo
2  // memorizzato in una variabile char*
3  #include <iostream.h>
4
5  int main()
6  {
7     char *string = "test";
8
9     cout << "Value of string is: " << string
10         << "\nValue of static_cast< void *>( string ) is: "
11         << static_cast< void *>( string ) << endl;
12    return 0;
13 }
```

```
Value of string is: test
Value of static_cast< void *>( string ) is: 0x00416D50
```

Figura 11.8 Visualizzazione dell'indirizzo memorizzato in una variabile di tipo char*.

11.3.4 L'output di caratteri tramite put

La funzione membro `put` invia in output un carattere:

```
cout.put( 'A' );
```

visualizza A sullo schermo. Le chiamate a `put` possono essere scritte in cascata:

```
cout.put( 'A' ).put( '\n' );
```


invia in output la lettera A seguita dal carattere di nuova linea. Come per `<<`, l'istruzione precedente ha questo effetto perché l'operatore punto associa da sinistra a destra e la funzione membro `put` restituisce un riferimento all'oggetto sulla quale è invocata. La funzione `put` può anche essere chiamata con un'espressione numerica che abbia un valore ASCII valido, come per es. `cout.put(65)`, che invia ugualmente in output il carattere A.

11.4 L'input da stream

Passiamo ora all'input da stream che viene effettuato per mezzo dell'operatore di estrazione dallo stream. Questo operatore normalmente ignora i caratteri di spaziatura presenti nello stream di input cioè spazi, tabulazioni e caratteri di nuova linea ma vedremo in seguito come modificare questo comportamento. L'operatore `>>` restituisce un riferimento nullo (verificabile come condizione falsa) se incontra la fine del file su uno stream, altrimenti restituisce un riferimento all'oggetto sul quale è stato invocato. Ogni stream contiene un insieme di *bit di stato* che servono a controllare lo stato dello stream, per esempio la formattazione, lo stato degli errori e così via. L'operazione di estrazione imposta a uno il `failbit` se si riscontra un valore del tipo errato in input, e imposta a uno il `badbit` se l'operazione non va a buon fine. Vedremo tra breve come controllare il valore di questi flag dopo un'operazione di I/O. Le Sezioni 11.7 e 11.8 trattano in dettaglio dei bit di stato degli stream.

11.4.1 L'operatore di estrazione dallo stream

Per leggere due interi, utilizzate l'oggetto `cin` e l'operatore `>>` come nella Figura 11.9. Osservate come sia possibile scrivere in cascata le operazioni di estrazione.

```
1  // Fig. 11.9: fig11_09.cpp
2  // Calcolo della somma di due interi immessi alla tastiera e
3  // rilevati in input con cin e l'operatore di estrazione
4  // dallo stream.
5  #include <iostream.h>
6
7  int main()
8  {
9     int x, y;
10
11    cout << "Enter two integers: ";
12    cin >> x >> y;
13    cout << "Sum of " << x << " and " << y << " is: "
14         << ( x + y ) << endl;
15
16    return 0;
17 }
```

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Figura 11.9 Calcolo della somma di due interi letti dalla tastiera tramite cin e l'operatore di estrazione dallo stream.

La precedenza piuttosto alta degli operatori >> e << può causare qualche problema. Per esempio, il programma nella Figura 11.10 non potrà essere compilato correttamente senza racchiudere tra parentesi l'espressione condizionale: provate a verificarlo voi stessi.

```
1   // Fig. 11.10: fig11_10.cpp
2   // Attenzione ai problemi di precedenza tra l'operatore di
3   // inserimento nello stream e l'operatore condizionale! Ci vuole
4   // una coppia di parentesi intorno all'espressione condizionale.
5   #include <iostream.h>
6
7   int main()
8   {
9      int x, y;
10
11     cout << "Enter two integers: ";
12     cin >> x >> y;
13     cout << x << ( x == y ? " is" : " is not" )
14          << " equal to " << y << endl;
15
16     return 0;
17  }
```

```
Enter two integers: 7 5
7 is not equal to 5
Enter two integers: 8 8
8 is equal to 8
```

Figura 11.10 Problema di precedenza evitato tra l'operatore di inserimento nello stream e l'operatore condizionale.

Errore tipico 11.1

Se tentate di leggere da un oggetto `ostream` *o da qualsiasi altro stream di solo output commettete un errore.*

Errore tipico 11.2

Se tentate di scrivere in un oggetto `istream` *o in qualsiasi altro stream di solo input commettete un errore.*

Errore tipico 11.3

Fate attenzione ai problemi che può causare l'uso degli operatori << e >> assieme ad altri operatori. La precedenza di << e >> è alta, per cui spesso si deve ricorrere alle parentesi per forzare il giusto ordine di valutazione di un'espressione che li contiene. Non dimenticatevene o commetterete degli errori.

Un modo tipico di effettuare l'input di una serie di valori consiste nell'utilizzare l'operazione di estrazione dallo stream all'interno della condizione di un ciclo `while`. L'estrazione restituisce un valore nullo (verificabile come condizione falsa) quando viene incontrata la fine del file. Osserviamo il programma nella Figura 11.11, che stabilisce qual è il voto massimo di un esame. Supponiamo che il numero di voti non sia noto in anticipo e che si

debba verificare la condizione "fine del file" per sapere quando interrompere l'input dei valori. La condizione del `while`, cioè `( cin >> grade )`, diventa falsa quando l'utente digita il carattere di fine del file (<ctrl>-d sui sistemi UNIX e Macintosh o <ctrl>-z sui sistemi MS-DOS, Windows e VMS).

```
1   // Fig. 11.11: fig11_11.cpp - L'operatore di estrazione
2   // restituisce false quando incontra la fine del file.
3   #include <iostream.h>
4
5   int main()
6   {
7      int grade, highestGrade = -1;
8
9      cout << "Enter grade (enter end-of-file to end): ";
10     while ( cin >> grade ) {
11        if ( grade > highestGrade )
12           highestGrade = grade;
13
14        cout << "Enter grade (enter end-of-file to end): ";
15     }
16
17     cout << "\n\nHighest grade is: " << highestGrade << endl;
18     return 0;
19  }
```

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 80
Enter grade (enter end-of-file to end): 33
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z

Highest grade is: 99
```

Figura 11.11 Operatore di estrazione dallo stream che restituisce false quando incontra la fine del file.

Obiettivo portabilità 11.1

Quando indicate all'utente come digitare la fine del file da tastiera, usate il messaggio "digitare il carattere di fine del file per terminare l'input" anziché indicargli di digitare lo specifico carattere di fine del file per il sistema sul quale state sviluppando (<ctrl>-d per UNIX e Macintosh e <ctrl>-z per PC e VMS).

Nella Figura 11.11, `cin >> grade` può essere utilizzato come condizione perché la classe base `ios` (da cui deriva `istream`) fornisce l'overloading di un operatore di cast che converte uno stream in un puntatore di tipo `void *` utilizzabile implicitamente dal compilatore. Il valore del puntatore è `0` se si è verificato un errore nella lettura di un valore o se è stata incontrato l'indicatore di fine del file.

11.4.2 Le funzioni membro get e getline

La funzione membro **get** senza argomenti effettua l'input di un carattere dallo stream indicato (anche di un carattere di spaziatura) e restituisce il carattere letto. Questa versione di **get** restituisce il carattere **EOF** quando incontra la fine del file sullo stream.

Il programma nella Figura 11.12 mostra l'uso delle funzioni membro **eof** e **get** sullo stream di input **cin** e della funzione membro **put** sullo stream di output **cout**. All'inizio il programma visualizza il valore di **cin.eof()**, che è **false** (**0** nell'output) per mostrare che non è stata ancora incontrata la fine del file su **cin**. L'utente digita una linea di testo e preme Invio seguito dal carattere di fine del file (<ctrl>-z sui sistemi MS-DOS, Windows e VMS, <ctrl>-d sui sistemi UNIX e Macintosh). Il programma legge ogni carattere e lo invia in output su **cout** tramite la funzione membro **put**. Quando viene incontrata la fine del file, il ciclo **while** termina e **cin.eof()**, che ora vale **true**, viene visualizzato nuovamente (**1** nell'output) per mostrare che ora la fine del file è stata raggiunta su **cin**. Osservate che questo programma utilizza la versione di **get** che non prende argomenti e restituisce il carattere letto in input.

La funzione membro **get** con un argomento di tipo carattere effettua l'input del carattere successivo presente nello stream di input (anche se è un carattere di spaziatura) e lo memorizza nell'argomento. Questa versione di **get** restituisce **0** quando incontra la fine del file, altrimenti restituisce un riferimento all'oggetto **istream** per cui è stata invocata.

```
1   // Fig. 11.12: fig11_12.cpp
2   // Utilizzo delle funzioni membro get, put ed eof.
3   #include <iostream.h>
4
5   int main()
6   {
7      char c;
8
9      cout << "Before input, cin.eof() is " << cin.eof()
10          << "\nEnter a sentence followed by end-of-file:\n";
11
12     while ( ( c = cin.get() ) != EOF)
13        cout.put( c );
14
15     cout << "\nEOF in this system is: " << c;
16     cout << "\nAfter input, cin.eof() is " << cin.eof() << endl;
17     return 0;
18  }
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
EOF in this system is: -1
After input cin.eof() is 1
```

Figura 11.12 Uso delle funzioni membro get, put e eof.

Una terza versione di **get** prende tre argomenti: un array di caratteri, un valore che esprime un limite dimensionale e un delimitatore (che per default vale **'\n'**). Questa versione legge un numero massimo di caratteri pari al limite specificato meno uno e termina, oppure termina non appena incontra in input il delimitatore. Per terminare la stringa di input viene aggiunto un carattere nullo nell'array di caratteri utilizzato come buffer. Il carattere delimitatore non viene trascritto nell'array, ma rimane comunque nello stream di input e sarà il carattere successivo a essere restituito da un'operazione di lettura. Così il risultato di un secondo **get** consecutivo è una linea vuota, a meno che il delimitatore non venga espulso dallo stream di input con un'operazione di flush. Il programma nella Figura 11.13 confronta l'input tramite **cin** con l'estrazione (che legge i caratteri finché non ne incontra uno di spaziatura) e l'input tramite **cin.get**. Osservate che la chiamata a **cin.get** non specifica un carattere delimitatore, per cui viene utilizzato il carattere di default **'\n'** La funzione membro **getline** opera come la terza versione di **get** e inserisce un carattere nullo dopo la linea letta nell'array di caratteri. La funzione **getline** elimina il delimitatore dallo stream, ovvero lo legge e lo scarta, senza memorizzarlo nell'array. Il programma nella Figura 11.14 mostra l'uso della funzione **getline** per l'input di una linea di testo.

```
1   // Fig. 11.13: fig11_13.cpp
2   // Confronto tra l'input di una stringa con cin e cin.get.
3   #include <iostream.h>
4
5   int main()
6   {
7      const int SIZE = 80;
8      char buffer1[ SIZE ], buffer2[ SIZE ];
9
10     cout << "Enter a sentence:\n";
11     cin >> buffer1;
12     cout << "\nThe string read with cin was:\n"
13          << buffer1 << "\n\n";
14
15     cin.get( buffer2, SIZE );
16     cout << "The string read with cin.get was:\n"
17          << buffer2 << endl;
18
19     return 0;
20  }
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Figura 11.13 Confronto delle operazioni di input con estrazione dallo stream e con la funzione cin.get.

```
1   // Fig. 11.14: fig11_14.cpp
2   // Input di un carattere con la funzione membro getline.
3   #include <iostream.h>
4
5   int main()
6   {
7      const SIZE = 80;
8      char buffer[ SIZE ];
9
10     cout << "Enter a sentence:\n";
11     cin.getline( buffer, SIZE );
12
13     cout << "\nThe sentence entered is:\n" << buffer << endl;
14     return 0;
15  }
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Figura 11.14 Input di caratteri tramite la funzione membro **getline**.

11.4.3 Altre funzioni membro di istream: peek, putback e ignore

La funzione membro `ignore` salta un determinato numero di caratteri (per default uno) o termina quando incontra il delimitatore indicato (per default `EOF` e, in questo caso, `ignore` salta alla fine del file).

La funzione membro `putback` prende il carattere ottenuto in precedenza da `get` da uno stream di input e lo pone nuovamente nello stream. Questa funzione è utile per le applicazioni che esaminano uno stream di input alla ricerca di un campo che inizia per un determinato carattere; quando esso viene rilevato nell'input, l'applicazione lo ripone nuovamente nello stream, in modo che possa ancora far parte dei dati che occorre leggere da quel momento in poi.

La funzione membro `peek` restituisce il carattere successivo presente nello stream di input, senza eliminarlo dallo stream.

11.4.4 L'I/O type-safe

Il C++ offre funzionalità di I/O che sono *type-safe* (cioè garantiscono la correttezza di tipo): grazie all'overloading, gli operatori `<<` e `>>` sono in grado di accettare tipi specifici e, se vengono elaborati tipi di dati inattesi, il sistema di I/O imposta vari flag di errore accessibili dal programmatore. In questo modo il programma può restare sotto il controllo dell'utente, non generandosi degli errori fatali di esecuzione. Riparleremo dei flag di errore nella Sezione 11.8.

11.5 L'input/output non formattato delle funzioni read, gcount e write

Le funzioni membro `read` e `write` effettuano l'I/O non formattato. Ognuna di esse si occupa del trasferimento di un certo numero di byte da/a un array di caratteri che si trova in memoria. Questi byte non sono formattati in alcun modo, ma sono semplicemente trattati come byte "grezzi". Per esempio, la chiamata

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write( buffer, 10 );
```

invia in output i primi 10 byte di `buffer`, incluso il carattere nullo che terminerebbe un'istruzione di output con `cout` e `<<`. Dato che una stringa di caratteri ha il valore dell'indirizzo del suo primo carattere, la chiamata

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10 );
```

invia in output le prime 10 lettere dell'alfabeto.

La funzione membro `read` riceve in input un determinato numero di caratteri e li memorizza in un array di caratteri. Inoltre, se in lettura riscontra un numero di caratteri inferiore a quello previsto, attiva il `failbit`. Vedremo tra breve come determinare se il `failbit` è stato attivato (cfr. Sezione 11.8).

La funzione membro `gcount` riporta il numero di caratteri letto dall'ultima operazione di input.

Il programma nella Figura 11.15 mostra le funzioni membro di `istream` `read` e `gcount` e la funzione membro di `ostream` `write`. Il programma legge dall'input 20 caratteri (da una sequenza di input più lunga) e li memorizza nell'array di caratteri `buffer` con `read`; determina il numero di caratteri letti con `gcount` e invia in output i caratteri in `buffer` con `write`.

11.6 I manipolatori di stream

In C++ esistono diversi *manipolatori di stream*, il cui compito è controllare la formattazione delle operazioni di I/O. I manipolatori di stream offrono numerose funzionalità: impostano l'ampiezza dei campi, la precisione, i flag di formattazione e i caratteri di riempimento dei campi, svuotano (flush) gli stream, inseriscono un carattere di nuova linea in uno stream di output e lo svuotano, inseriscono un carattere nullo in uno stream di output stream e saltano i caratteri di spaziatura in uno stream di input. Tutte queste funzionalità sono descritte nelle sezioni che seguono.

```
1   // Fig. 11.15: fig11_15.cpp
2   // I/O non formattato con read, gcount e write.
3   #include <iostream.h>
4
5   int main()
6   {
7      const int SIZE = 80;
8      char buffer[ SIZE ];
```

Figura 11.15 I/O non formattato tramite le funzioni membro read, gcount e write (continua).

```
 9
10       cout << "Enter a sentence:\n";
11       cin.read( buffer, 20 );
12       cout << "\nThe sentence entered was:\n";
13       cout.write( buffer, cin.gcount() );
14       cout << endl;
15       return 0;
16    }
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Figura 11.15 I/O non formattato tramite le funzioni membro read, gcount e write.

11.6.1 La base dei numeri interi su uno stream: dec, oct, hex e setbase

Gli interi normalmente sono visualizzati come valori decimali. Per cambiare la base in cui sono interpretati su un determinato stream sono disponibili alcuni manipolatori: `hex` per la base esadecimale (base 16), `oct` per la base ottale (base 8) e `dec` che reimposta la base decimale. La base di uno stream può anche essere modificata tramite il manipolatore `setbase` che prende un argomento intero, a scelta tra 10, 8 e 16, e imposta la base corrispondente. Dato che `setbase` prevede un argomento, è detto *manipolatore di stream parametrizzato*. Se si desidera utilizzare `setbase` o qualsiasi altro manipolatore di stream parametrizzato, bisogna includere nel programma il file di intestazione `<iomanip.h>`. La base dello stream rimane inalterata finché non è cambiata esplicitamente. Il programma nella Figura 11.16 mostra l'uso dei manipolatori `hex`, `oct`, `dec` e `setbase`.

11.6.2 La precisione dei valori a virgola mobile: precision e setprecision

È possibile controllare la precisione dei valori a virgola mobile, cioè il numero di cifre che compare a destra del punto decimale, utilizzando il manipolatore di stream `setprecision` o la funzione membro `precision`. Entrambi impostano la precisione per tutte le successive operazioni di output, fino a una successiva modifica. La funzione membro `precision` senza argomenti restituisce l'impostazione corrente della precisione. Il programma nella Figura 11.17 utilizza `precision` e `setprecision` per l'output di una tabella che visualizza la radice quadrata di 2 con una precisione che varia da 0 a 9 cifre decimali.

```
1    // Fig. 11.16: fig11_16.cpp
2    // Utilizzo dei manipolatori di stream hex, oct, dec e setbase.
3    #include <iostream.h>
4    #include <iomanip.h>
5
```

Figura 11.16 Uso dei manipolatori di stream `hex`, `oct`, `dec` e `setbase` (continua).


```
 6    int main()
 7    {
 8       int n;
 9
10       cout << "Enter a decimal number: ";
11       cin >> n;
12
13       cout << n << " in hexadecimal is: "
14            << hex << n << '\n'
15            << dec << n << " in octal is: "
16            << oct << n << '\n'
17            << setbase( 10 ) << n << " in decimal is: "
18            << n << endl;
19
20       return 0;
21    }
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Figura 11.16 Uso dei manipolatori di stream `hex`, `oct`, `dec` e `setbase`.

```
1    // Fig. 11.17: fig11_17.cpp - Controllo della precisione
2    // nella visualizzazione dei valori a virgola mobile
3    #include <iostream.h>
4    #include <iomanip.h>
5    #include <math.h>
6
7    int main()
8    {
9       double root2 = sqrt( 2.0 );
10      int places;
11
12      cout << setiosflags( ios::fixed)
13           << "Square root of 2 with precisions 0-9.\n"
14           << "Precision set by the "
15           << "precision member function:" << endl;
16
17      for ( places = 0; places <= 9; places++ ) {
18         cout.precision( places );
19         cout << root2 << '\n';
20      }
21
22      cout << "\nPrecision set by the "
23           << "setprecision manipulator:\n";
24
```

Figura 11.17 Controllo della precisione dei valori a virgola mobile (continua).

```
25       for ( places = 0; places <= 9; places++ )
26          cout << setprecision( places ) << root2 << '\n';
27
28       return 0;
29    }
```

```
Square root of 2 with precisions 0-9.
Precision set by the precision member function:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by the setprecision manipulator:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Figura 11.17 Controllo della precisione dei valori a virgola mobile.

11.6.3 L'ampiezza dei campi: setw e width

La funzione `width`, membro di `ios`, imposta l'ampiezza di un campo, cioè il numero di posizioni carattere in cui deve restare l'input o l'output di un valore, e restituisce l'ampiezza impostata precedentemente. Se i valori elaborati sono più piccoli dell'ampiezza specificata per il campo, questo viene completato con dei *caratteri di riempimento*. Se un valore è più esteso dell'ampiezza designata viene visualizzato per intero, senza alcun troncamento.

Errore tipico 11.4

L'impostazione dell'ampiezza del campo vale solo per la successiva operazione di inserimento o estrazione, dopo di che torna implicitamente a `0`, vale a dire ogni valore in output occuperà tutta l'ampiezza necessaria. La funzione `width` senza argomenti restituisce l'impostazione corrente. Se date per scontato che una chiamata a `width` si applichi a tutti gli output da quel momento in poi, commettete un errore logico.

Errore tipico 11.5

Se non prevedete un'ampiezza sufficiente per l'output dei valori, questi occuperanno tutto lo spazio di cui hanno bisogno, anche oltre il limite del campo che avete indicato: in questo modo l'output potrebbe diventare poco leggibile.

Il programma nella Figura 11.18 mostra l'uso della funzione membro `width` per l'input e per l'output. Osservate che, per quanto riguarda l'input, sarà letto un numero massimo di caratteri inferiore di uno rispetto all'ampiezza del campo, perché è previsto anche lo spazio per il terminatore nullo della stringa. Tenete a mente che l'estrazione dallo stream termina quando si incontra un carattere di spaziatura (che non sia iniziale). Si può utilizzare anche il manipolatore `setw` per impostare l'ampiezza di un campo. *Nota*: Quando si richiede un input all'utente, questi dovrebbe digitare una linea di testo e premere Invio seguito dal carattere di fine del file (<ctrl>-z sui sistemi MS-DOS, Windows o VMS, <ctrl>-d sui sistemi UNIX e Macintosh).

```
1   // fig11_18.cpp
2   // Uso della funzione membro width
3   #include <iostream.h>
4
5   int main()
6   {
7      int w = 4;
8      char string[ 10 ];
9
10     cout << "Enter a sentence:\n";
11     cin.width( 5 );
12
13     while ( cin >> string ) {
14        cout.width( w++ );
15        cout << string << endl;
16        cin.width( 5 );
17     }
18
19     return 0;
20  }
```

```
Enter a sentence:
This is a test of the width function
This
  is
   a
  test
    of
   the
  widt
     h
  func
    tion
```

Figura 11.18 Uso della funzione membro width.

11.6.4 I manipolatori definiti dall'utente

È possibile creare dei propri manipolatori di stream. Il programma nella Figura 11.19 mostra come creare e utilizzare i nuovi manipolatori `bell` (beep), `ret` (a capo), `tab` (tabulazione) e `endLine` (*fine linea*). È possibile creare anche nuovi manipolatori di stream parametrizzati: consultate i manuali del vostro compilatore per sapere come.

```
1  // Fig. 11.19: fig11_19.cpp
2  // Creazione e verifica di un manipolatore di stream
3  // non parametrizzato definito dall'utente.
4  #include <iostream.h>
5
6  // manipolatore bell (che utilizza la sequenza di escape \a)
7  ostream& bell( ostream& output ) { return output << '\a'; }
8
9  // manipolatore ret (che utilizza la sequenza di escape \r)
10 ostream& ret( ostream& output ) { return output << '\r'; }
11
12 // manipolatore tab (che utilizza la sequenza di escape \t)
13 ostream& tab( ostream& output ) { return output << '\t'; }
14
15 // manipolatore endLine (che utilizza la sequenza di escape \n
16 // e la funzione membro flush)
17 ostream& endLine( ostream& output )
18 {
19    return output << '\n' << flush;
20 }
21
22 int main()
23 {
24    cout << "Testing the tab manipulator:" << endLine
25         << 'a' << tab << 'b' << tab << 'c' << endLine
26         << "Testing the ret and bell manipulators:"
27         << endLine << ".........";
28    cout << bell;
29    cout << ret << "--" << endLine;
30    return 0;
31 }
```

```
Testing the tab manipulator:
a       b       c
Testing the ret and bell manipulators:
--
```

Figura 11.19 Creazione e verifica di manipolatori di stream definiti dall'utente e non parametrizzati.

11.7 I valori di stato della formattazione

Vari *flag di formattazione* specificano il tipo di formattazione da utilizzare nelle successive operazioni di I/O su stream. Le funzioni membro `setf`, `unsetf` e `flags` controllano lo stato dei flag.

11.7.1 I flag di stato della formattazione

Ogni flag di stato della formattazione nella Figura 11.20 (insieme con altri che non sono in figura) è definito come un'enumerazione nella classe `ios`. La loro descrizione è presentata nelle sezioni che seguono.

Flag di stato della formattazione	Descrizione
`ios::skipws`	Salta i caratteri di spaziatura su uno stream di input.
`ios::left`	Giustifica a sinistra l'output in un campo. I caratteri di riempimento, se necessari, compaiono a destra.
`ios::right`	Giustifica a destra l'output in un campo. I caratteri di riempimento, se necessari, compaiono a sinistra.
`ios::internal`	Indica che il segno di un numero deve essere giustificato a sinistra in un campo mentre il suo valore deve essere giustificato a destra nello stesso campo (cioè possono comparire caratteri di riempimento tra il segno e il valore).
`ios::dec`	Specifica che gli interi devono essere trattati in base 10.
`ios::oct`	Specifica che gli interi devono essere trattati come valori ottali (base 8).
`ios::hex`	Specifica che gli interi devono essere trattati come valori esadecimali (base 16).
`ios::showbase`	Specifica che la base di un numero deve essere indicata nell'output prima del suo valore (si ha uno `0` iniziale per i numeri ottali e il simbolo `0x` o `0X` per gli esadecimali).
`ios::showpoint`	Specifica che i numeri a virgola mobile devono essere inviati in output con il punto decimale. Questa opzione viene normalmente utilizzata in coppia con `ios::fixed` per fissare un certo numero di cifre a destra del punto decimale.
`ios::uppercase`	Specifica di utilizzare la lettera maiuscola `X` nel simbolo `0X` che prefissa un numero esadecimale e la lettera maiuscola `E` nei valori a virgola mobile in notazione scientifica.
`ios::showpos`	Specifica che gli interi positivi e negativi devono essere prefissati esplicitamente dal proprio segno.
`ios::scientific`	Specifica l'output di un valore a virgola mobile nella notazione scientifica.
`ios::fixed`	Specifica l'output di un valore a virgola mobile con un determinato numero di cifre a destra del punto decimale.

Figura 11.20 Flag di stato della formattazione.

Questi flag sono controllati dalle funzioni membro `flags`, `setf` e `unsetf`, ma molti programmatori preferiscono utilizzare i manipolatori di stream (cfr. Sezione 11.7.8). È possibile utilizzare l'operatore OR sui bit, `|`, per combinare diverse opzioni (cfr. Fig. 11.23). Quando si chiama `flags` per un dato stream specificando diverse opzioni unite dagli "OR", la funzione imposta tutte le opzioni specificate su quel determinato stream e restituisce un valore `long` che contiene le opzioni precedenti.

Di norma questo valore viene salvato, in modo che `flags` possa essere nuovamente chiamata in seguito per ripristinare le opzioni precedenti dello stream.

La funzione `flags` specifica un valore che rappresenta le impostazioni di tutti i flag. La funzione `setf` che prende un argomento, d'altra parte, specifica uno o più flag nel formato "OR" ed effettua un "OR" tra le nuove opzioni specificate e i flag già impostati, creando un nuovo stato della formattazione.

Il manipolatore di stream parametrizzato `setiosflags` effettua le stesse operazioni della funzione membro `setf`. Il manipolatore `resetiosflags` effettua le stesse operazioni della funzione membro `unsetf`. Per utilizzare ciascuno di questi manipolatori dovete includere il file di intestazione `<iomanip.h>`.

Il flag `skipws` indica che `>>` deve saltare i caratteri di spaziatura in uno stream di input. Il comportamento di default di `>>` lo prevede già. Per cambiare questa impostazione, scrivete `unsetf(ios::skipws)`. Per specificare che si desidera saltare i caratteri di spaziatura si può anche utilizzare il manipolatore `ws`.

11.7.2 Gli zero in coda e i punti decimali: ios::showpoint

Il flag `showpoint` viene attivato per forzare la visualizzazione di un numero a virgola mobile con il punto decimale e diverse cifre zero in coda ai decimali. Se `showpoint` non è attivo, un valore come `79.0` è visualizzato come `79`, mentre se `showpoint` è attivo, esso viene visualizzato come `79.000000`: il numero di zero è specificato dalla precisione corrente. Il programma nella Figura 11.21 mostra come impostare il flag `showpoint` con la funzione membro `setf`.

```
1   // FIG. 11.21: fig11_21.cpp
2   // Controllo della visualizzazione degli zero in coda e
3   // dei punti decimali per i valori a virgola mobile.
4   #include <iostream.h>
5   #include <iomanip.h>
6   #include <math.h>
7
8   int main()
9   {
10     cout << "Before setting the ios::showpoint flag\n"
11          << "9.9900 prints as: " << 9.9900
12          << "\n9.9000 prints as: " << 9.9000
13          << "\n9.0000 prints as: " << 9.0000
14          << "\n\nAfter setting the ios::showpoint flag\n";
```

Figura 11.21 Controllo del punto decimale e degli zero in coda ai valori a virgola mobile (continua).


```
15     cout.setf( ios::showpoint );
16     cout << "9.9900 prints as: " << 9.9900
17          << "\n9.9000 prints as: " << 9.9000
18          << "\n9.0000 prints as: " << 9.0000 << endl;
19     return 0;
20  }
```

```
Before setting the ios::showpoint flag
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After setting the ios::showpoint flag
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Figura 11.21 Controllo del punto decimale e degli zero in coda ai valori a virgola mobile.

11.7.3 La giustificazione: ios::left, ios::right e ios::internal

I flag `left` e `right` controllano rispettivamente la giustificazione a sinistra (con caratteri di riempimento a destra) e la giustificazione a destra (con caratteri di riempimento a sinistra). Il carattere da utilizzare per il riempimento del campo è specificato dalla funzione membro `fill` o dal manipolatore di stream parametrizzato `setfill` (cfr. Sezione 11.7.4). Il programma in Fig. 11.22 mostra l'uso dei manipolatori `setw`, `setiosflags` e `resetiosflags` e delle funzioni membro `setf` e `unsetf` per controllare la giustificazione a sinistra e a destra dei dati interi in un campo.

```
1   // Fig. 11.22: fig11_22.cpp
2   // Giustificazione a sinistra e a destra.
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      int x = 12345;
9
10     cout << "Default is right justified:\n"
11          << setw(10) << x << "\n\nUSING MEMBER FUNCTIONS"
12          << "\nUse setf to set ios::left:\n" << setw(10);
13
14     cout.setf( ios::left, ios::adjustfield );
15     cout << x << "\nUse unsetf to restore default:\n";
16     cout.unsetf( ios::left );
17     cout << setw( 10 ) << x
18          << "\n\nUSING PARAMETERIZED STREAM MANIPULATORS"
```

Figura 11.22 Giustificazione a sinistra e a destra (continua).

```
19          << "\nUse setiosflags to set ios::left:\n"
20          << setw( 10 ) << setiosflags( ios::left ) << x
21          << "\nUse resetiosflags to restore default:\n"
22          << setw( 10 ) << resetiosflags( ios::left )
23          << x << endl;
24       return 0;
25    }
```

```
Default is right justified:
     12345
USING MEMBER FUNCTIONS
Use setf to set ios::left:
12345
Use unsetf to restore default:
     12345
USING PARAMETERIZED STREAM MANIPULATORS
Use setiosflags to set ios::left:
12345
Use resetiosflags to restore default:
     12345
```

Figura 11.22 Giustificazione a sinistra e a destra.

Il flag `internal` indica che il segno di un numero (o la base quando è attivo il flag `ios::showbase`; cfr. Sezione 11.7.5) deve essere giustificato a sinistra in un campo, il valore del numero deve essere giustificato a destra e, se necessario, gli spazi che separano segno e numero devono essere riempiti con il carattere di riempimento. I flag `left`, `right` e `internal` sono contenuti nel dato membro statico `ios::adjustfield`. `ios::adjustfield` deve essere usato come secondo argomento di `setf` quando si impostano i flag di giustificazione. Ciò consente a `setf` di assicurarsi che sia stato attivato uno solo dei tre flag di giustificazione che si escluderebbero a vicenda. Il programma nella Figura 11.23 mostra l'uso dei manipolatori `setiosflags` e `setw` per specificare la formattazione `internal`. Osservate l'uso del flag `ios::showpos` per forzare la visualizzazione del segno +.

```
1   // Fig. 11.23: fig11_23.cpp
2   // Visualizzazione di un intero con spaziatura interna
3   // e con il segno più.
4   #include <iostream.h>
5   #include <iomanip.h>
6
7   int main()
8   {
9      cout << setiosflags( ios::internal | ios::showpos )
10          << setw( 10 ) << 123 << endl;
11     return 0;
12  }
```

```
+       123
```

Figura 11.23 Visualizzazione di un intero con formattazione internal e con il segno +.

11.7.4 Il riempimento dei campi: fill e setfill

La funzione membro `fill` specifica il carattere di riempimento da utilizzare nei campi giustificati e restituisce il carattere di riempimento impostato precedentemente. Se non viene specificato alcun valore per il carattere di riempimento per default verrà utilizzato il carattere spazio. Il manipolatore parametrizzato `setfill` ha la stessa funzione di `fill`.

Il programma nella Figura 11.24 mostra l'uso della funzione membro `fill` e del manipolatore `setfill`.

```
1   // Fig. 11.24: fig11_24.cpp
2   // Utilizzo della funzione membro fill e del manipolatore setfill
3   // per modificare il carattere di riempimento per campi
4   // più estesi dei valori da visualizzare.
5   #include <iostream.h>
6   #include <iomanip.h>
7
8   int main()
9   {
10     int x = 10000;11
11
12     cout << x << " printed as int right and left justified\n"
13          << "and as hex with internal justification.\n"
14          << "Using the default pad character (space):\n";
15     cout.setf( ios::showbase );
16     cout << setw( 10 ) << x << '\n';
17     cout.setf( ios::left, ios::adjustfield );
18     cout << setw( 10 ) << x << '\n';
19     cout.setf( ios::internal, ios::adjustfield );
20     cout << setw( 10 ) << hex << x;
21
22     cout << "\n\nUsing various padding characters:\n";
23     cout.setf( ios::right, ios::adjustfield );
24     cout.fill( '*' );
25     cout << setw( 10 ) << dec << x << '\n';
26     cout.setf( ios::left, ios::adjustfield );
27     cout << setw( 10 ) << setfill( '%' ) << x << '\n';
28     cout.setf( ios::internal, ios::adjustfield );
29     cout << setw( 10 ) << setfill( '^' ) << hex << x << endl;
30     return 0;
31  }
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x     2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Figura 11.24 Uso di fill e setfill per modificare il carattere di riempimento di campi più estesi dei valori che devono contenere durante l'output (continua).

```
Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Figura 11.24 Uso di fill e setfill per modificare il carattere di riempimento di campi più estesi dei valori che devono contenere durante l'output.

11.7.5 La base dei numeri interi: ios::dec, ios::oct, ios::hex e ios::showbase

Il membro statico `ios::basefield` (utilizzato in modo simile a `ios::adjustfield` per `setf`) include i bit di flag `ios::oct`, `ios::hex` e `ios::dec`, che specificano rispettivamente che gli interi devono essere trattati come valori ottali, esadecimali o decimali. Se non è attivo nessuno di questi bit, per default le operazioni di inserimento nello stream visualizzano un valore decimale. Il default per le estrazioni dallo stream è di elaborare i dati nella forma in cui sono forniti: gli interi che iniziano per `0` sono trattati come valori ottali, quelli che iniziano per `0x` o `0X` come valori esadecimali e tutti gli altri come valori decimali. Quando si specifica una determinata base per uno stream, tutti gli interi di quello stream sono elaborati in quella base, finché non se ne specifica una nuova o fino al termine del programma. Attivate il flag `showbase` per forzare la visualizzazione della base di un valore intero. I numeri decimali saranno visualizzati normalmente, gli ottali saranno prefissati con uno `0` e gli esadecimali con il simbolo `0x` o `0X` (il flag `uppercase` determina quale dei due; cfr. Sezione 11.7.7). Il programma nella Figura 11.25 mostra l'uso del flag `showbase`.

```
1   // Fig. 11.25: fig11_25.cpp
2   // Utilizzo del flag ios::showbase
3   #include <iostream.h>
4   #include <iomanip.h>
5
6   int main()
7   {
8      int x = 100;
9
10     cout << setiosflags( ios::showbase )
11          << "Printing integers preceded by their base:\n"
12          << x << '\n'
13          << oct << x << '\n'
14          << hex << x << endl;
15     return 0;
16  }
```

```
Printing integers preceded by their base:
100
0144
0x64
```

Figura 11.25– Uso del flag ios::showbase.

11.7.6 I numeri a virgola mobile e la notazione scientifica: ios::scientific e ios::fixed

I flag `ios::scientific` e `ios::fixed` sono contenuti nel dato membro statico `ios::floatfield` (usato in modo simile a `ios::adjustfield` e `ios::basefield` in `setf`) e controllano la formattazione dell'output dei numeri a virgola mobile.

Il flag `scientific` forza l'output nel formato scientifico (in notazione esponenziale), mentre il flag `fixed` forza la visualizzazione di un numero fisso di cifre a destra del punto decimale: il numero esatto delle cifre si stabilisce con la funzione membro `precision`. Se questi flag non sono attivi, è il valore del numero a determinarne la formattazione dell'output.

La chiamata `cout.setf(0, ios::floatfield)` ripristina la formattazione di default del sistema per i valori a virgola mobile. Il programma nella Figura 11.26 mostra la visualizzazione dei numeri a virgola mobile nei formati `fixed` e `scientific` utilizzando la versione a due argomenti di `setf` con `ios::floatfield`.

```
1   // Fig. 11.26: fig11_26.cpp
2   // Visualizzazione di valori a virgola mobile nei formati di
3   // default, scientifico e fisso.
4   #include <iostream.h>
5
6   int main()
7   {
8      double x = .001234567, y = 1.946e9;
9
10     cout << "Displayed in default format:\n"
11          << x << '\t' << y << '\n';
12     cout.setf( ios::scientific, ios::floatfield );
13     cout << "Displayed in scientific format:\n"
14          << x << '\t' << y << '\n';
15     cout.unsetf( ios::scientific );
16     cout << "Displayed in default format after unsetf:\n"
17          << x << '\t' << y << '\n';
18     cout.setf( ios::fixed, ios::floatfield );
19     cout << "Displayed in fixed format:\n"
20          << x << '\t' << y << endl;
21     return 0;
22  }
```

```
Displayed in default format:
0.00123457      1.946e+009
Displayed in scientific format:
1.234567e-003   1.946000e+009
Displayed in default format after unsetf:
0.00123457      1.946e+009
Displayed in fixed format:
0.001235        1946000000.000000
```

Figura 11.26 Visualizzazione di valori a virgola mobile nella formattazione di sistema, scientific e fixed.

11.7.7 Il controllo delle lettere maiuscole/minuscole: ios::uppercase

Il flag `ios::uppercase` forza la visualizzazione della `X` o della `E` maiuscola, rispettivamente per gli interi esadecimali e i valori a virgola mobile in formato scientifico (Fig. 11.27). Quando è attivo, il flag `ios::uppercase` causa la visualizzazione in maiuscolo di tutte le lettere di un valore esadecimale.

```
1   // Fig. 11.27: fig11_27.cpp - Utilizzo del flag ios::uppercase
2   #include <iostream.h>
3   #include <iomanip.h>
4
5   int main()
6   {
7      cout << setiosflags( ios::uppercase )
8         << "Printing uppercase letters in scientific\n"
9           << "notation exponents and hexadecimal values:\n"
10          << 4.345e10 << '\n' << hex << 123456789 << endl;
11     return 0;
12  }
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
75BCD15
```

Figura 11.27 Uso del flag ios::uppercase .

11.7.8 L'attivazione e la disattivazione dei flag di formattazione: flags, setiosflags e resetiosflags

La funzione membro `flags` senza argomenti restituisce semplicemente le impostazioni correnti dei flag di formattazioni, nella forma di un valore `long`. La funzione `flags` con un argomento di tipo `long` imposta i flag di formattazione, così come sono specificati nell'argomento, e restituisce le impostazioni precedenti. Tutti i flag di formattazione non specificati esplicitamente nell'argomento di `flags` sono disattivati. Notate che le impostazioni iniziali dei flag possono variare da sistema a sistema. Il programma nella Figura 11.28 usa di `flags` per impostare un nuovo stato della formattazione salvando quello precedente, e ripristina successivamente lo stato iniziale.

```
1   // Fig. 11.28: fig11_28.cpp
2   // Utilizzo della funzione membro flags.
3   #include <iostream.h>
4
5   int main()
6   {
7      int i = 1000;
8      double d = 0.0947628;
```

Figura 11.28 Uso della funzione membro flags (continua).


```
9
10     cout << "The value of the flags variable is: "
11          << cout.flags()
12          << "\nPrint int and double in original format:\n"
13          << i << '\t' << d << "\n\n";
14     long originalFormat =
15           cout.flags( ios::oct | ios::scientific );
16     cout << "The value of the flags variable is: "
17          << cout.flags()
18          << "\nPrint int and double in a new format\n"
19          << "specified using the flags member function:\n"
20          << i << '\t' << d << "\n\n";
21     cout.flags( originalFormat );
22     cout << "The value of the flags variable is: "
23          << cout.flags()
24          << "\nPrint values in original format again:\n"
25          << i << '\t' << d << endl;
26     return 0;
27  }
```

```
The value of the flags variable is: 0
Print int and double in original format:
1000    0.0947628

The value of the flags variable is: 4040
Print int and double in a new format
specified using the flags member function:
1750    9.476280e-002

The value of the flags variable is: 0
Print values in original format again:
1000    0.0947628
```

Figura 11.28 Uso della funzione membro flags.

La funzione membro `setf` attiva i flag di formattazione indicati dal suo argomento e restituisce le impostazioni precedenti dei flag in un valore `long`, come in

```
long previousFlagSettings =
      cout.setf( ios::showpoint | ios::showpos );
```

La funzione `setf` con due argomenti, come in

```
cout.setf( ios::left, ios::adjustfield );
```

azzera prima i bit di `ios::adjustfield` e poi attiva il flag `ios::left`. Questa versione di `setf` si usa con i campi bit associati a `ios::basefield` (rappresentato da `ios::dec`, `ios::oct` e `ios::hex`), `ios::floatfield` (rappresentato da `ios::scientific` e `ios::fixed`) e `ios::adjustfield` (rappresentato da `ios::left`, `ios::right` e `ios::internal`).

La funzione membro `unsetf` disattiva i flag designati e restituisce il valore dei flag prima dell'operazione.

11.8 I valori di stato degli errori in uno stream

Lo stato di uno stream può essere verificato grazie ai bit della classe `ios`, che è la classe base di tutte le classi di I/O da noi utilizzate (cioè `istream`, `ostream` e `iostream`).

Il bit `eofbit` è attivato automaticamente per uno stream di input se viene raggiunta la fine del file. Un programma può utilizzare la funzione membro `eof` per determinare questa condizione. La chiamata

```
cin.eof()
```

restituisce `true` se è stata incontrata la fine del file su `cin`, altrimenti restituisce `false`.

Il bit `failbit` è attivato per uno stream quando si verifica un errore di formattazione, senza perdita di caratteri. La funzione membro `fail` determina se un'operazione su stream non è riuscita: normalmente è possibile proseguire le operazioni dopo errori di questo genere.

Il bit `badbit` è attivato per uno stream quando si verifica un errore con perdita di dati. La funzione membro `bad` determina se un'operazione su stream non è riuscita. Normalmente non è possibile continuare le operazioni dopo errori di questa gravità.

Il bit `goodbit` è attivato per uno stream se non è stato attivato nessuno dei seguenti: `eofbit`, `failbit` e `badbit`. La funzione membro `good` restituisce `true` se tutte le funzioni `bad`, `fail` ed `eof` restituiscono `false`. Sarebbe lecito aspettarsi che tutte le operazioni di I/O dovrebbero essere "good" (in inglese "buone").

La funzione membro `rdstate` restituisce il valore di stato degli errori relativo a uno stream. Una chiamata a `cout.rdstate` restituisce lo stato di uno stream, il quale può essere poi esaminato in un'istruzione `switch`, per sapere quale condizione si è verificata tra `ios::eofbit`, `ios::badbit`, `ios::failbit` e `ios::goodbit`. Il modo preferito di verificare lo stato di uno stream consiste nell'utilizzare le funzioni membro `eof`, `bad`, `fail` e `good`: in questo modo il programmatore non deve occuparsi di gestire i particolari bit di stato.

La funzione membro `clear` si usa normalmente per ripristinare lo stato "good" di uno stream, in modo che le operazioni di I/O possano proseguire. L'argomento di default di `clear` è `ios::goodbit`, perciò l'istruzione

```
cin.clear();
```

ripristina lo stato "good" di `cin` e attiva il bit `goodbit` per lo stream. L'istruzione

```
cin.clear(ios::failbit)
```

attiva manualmente il `failbit`. Questa operazione può servire quando si verificano dei problemi durante l'input di tipi di dato definiti dall'utente. In questo contesto il nome `clear` può sembrare fuori luogo, ma è corretto.

Il programma nella Figura 11.29 illustra l'uso di `rdstate`, `eof`, `fail`, `bad`, `good` e `clear`.

La funzione membro `operator!` restituisce `true` se uno o entrambi i flag `badbit` e `failbit` sono attivi. La funzione membro `void*` restituisce `false` se uno o entrambi i bit `badbit` e `failbit` sono attivi. Queste funzioni sono utili per l'elaborazione dei file, quando si deve verificare una condizione `true`/`false` in una struttura di selezione o di iterazione.


```
1   // Fig. 11.29: fig11_29.cpp
2   // Verifica dei valori di stato degli errori.
3   #include <iostream.h>
4
5   int main()
6   {
7      int x;
8      cout << "Before a bad input operation:"
9           << "\ncin.rdstate(): " << cin.rdstate()
10          << "\n    cin.eof(): " << cin.eof()
11          << "\n   cin.fail(): " << cin.fail()
12          << "\n    cin.bad(): " << cin.bad()
13          << "\n   cin.good(): " << cin.good()
14          << "\n\nExpects an integer, but enter a character: ";
15     cin >> x;
16
17     cout << "\nEnter a bad input operation:"
18          << "\ncin.rdstate(): " << cin.rdstate()
19          << "\n    cin.eof(): " << cin.eof()
20          << "\n   cin.fail(): " << cin.fail()
21          << "\n    cin.bad(): " << cin.bad()
22          << "\n   cin.good(): " << cin.good() << "\n\n";
23
24     cin.clear();
25
26     cout << "After cin.clear()"
27          << "\ncin.fail(): " << cin.fail()
28          << "\ncin.good(): " << cin.good() << endl;
29     return 0;
30  }
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 2
    cin.bad(): 0
   cin.good(): 0

After cin.clear()
cin.fail(): 0
cin.good(): 1
```

Figura 11.29 Verifica dei valori di stato degli errori.

11.9 Il collegamento di uno stream di output a uno stream di input

Le applicazioni interattive hanno generalmente bisogno di un `istream` per l'input e di un `ostream` per l'output. Quando compare un messaggio sullo schermo, l'utente risponde digitando i dati appropriati. Naturalmente il "prompt", cioè il messaggio, deve comparire prima dell'operazione di input. Con il buffering, l'output compare soltanto quando il buffer si riempie oppure quando questo viene svuotato esplicitamente dal programma o automaticamente al termine del programma. Il C++ fornisce la funzione membro `tie` per sincronizzare (cioè collegare) l'operazione di un `istream` e di un `ostream`, assicurando che l'output compaia sempre prima dell'input successivo. La chiamata

```
cin.tie( &cout );
```

collega `cout` (un `ostream`) a `cin` (un `istream`). In realtà questa chiamata è ridondante, perché il C++ effettua automaticamente questa operazione, creando un ambiente standard di I/O per gli utenti, tuttavia un utente potrebbe desiderare di collegare esplicitamente una coppia `istream/ostream`. Per scollegare uno stream di input (ad esempio `cin`) da uno stream di output, utilizzate la chiamata

```
cin.tie( 0 );
```

Esercizi di autovalutazione

11.1 Completate le seguenti affermazioni:

a) Gli operatori degli stream in overloading sono spessi definiti come funzioni _____ di una classe.
b) I bit di giustificazione della formattazione che si possono attivare sono _____, _____ e _____.
c) L'input/output in C++ avviene per mezzo di _____ di byte.
d) I manipolatori parametrizzati degli stream _____ e _____ si usano per attivare e disattivare i flag di stato della formattazione.
e) I programmi C++ che usano le operazioni di I/O su stream devono includere il file di intestazione, _____ che contiene le informazioni di base necessarie,
f) Se si utilizzano manipolatori di stream parametrizzati, si deve includere il file di intestazione _____.
g) Il file di intestazione _____ contiene le informazioni per l'elaborazione dei file controllata dall'utente.
h) La funzione membro di `ostream` è utilizzata nell'output non formattato.
i) Le operazioni di input sono supportate dalla classe _____.
l) L'output dello stream di errore standard è diretto all'oggetto stream _____ o _____.
m) Le operazioni di output sono supportate dalla classe _____.
n) Il simbolo dell'operatore di inserimento nello stream è _____.
o) I quattro oggetti che corrispondono ai dispositivi standard del sistema sono _____, _____, _____ e _____.
p) Il simbolo dell'operatore di estrazione dallo stream è _____.
q) I manipolatori di stream _____, _____ e _____ servono a indicare che gli interi devono essere visualizzati in formato ottale, esadecimale e decimale.
r) La precisione di default per i valori a virgola mobile è _____.
s) Quando è attivo, il flag _____ causa la visualizzazione del segno "+" per i numeri interi positivi.

11.2 Vero/Falso. Per le affermazioni false, datene una breve spiegazione.

a) La funzione membro `flags()` con un argomento `long` imposta la variabile di stato `flags` al valore dell'argomento e restituisce il suo valore precedente.
b) Gli operatori di inserimento nello stream `<<` e di estrazione dallo stream `>>` sono in overloading per gestire tutti i tipi di dato standard, tra cui stringhe e indirizzi di memoria (solo `<<`), e tutti i tipi di dati definiti dall'utente.
c) La funzione membro `flags()` senza argomenti disattiva tutti i bit della variabile di stato `flags`.
d) L'operatore di estrazione dallo stream `>>` può subire overloading da una funzione operatore che prende come argomenti un riferimento a `istream` e un riferimento a un tipo definito dall'utente e restituisce un riferimento a `istream`.
e) Il manipolatore di stream `ws` salta i caratteri di spaziatura iniziali in uno stream di input.
f) L'operatore di inserimento nello stream `<<` può subire overloading da una funzione operatore che prende come argomenti un riferimento a `istream` e un riferimento a un tipo definito dall'utente e restituisce un riferimento a `istream`.
g) L'input effettuato con l'operatore `>>` salta sempre i caratteri di spaziatura iniziali in uno stream di input.
h) Le funzionalità di input e output sono parte integrante del C++.
i) La funzione membro `rdstate()` restituisce lo stato dello stream corrente.
j) Lo stream `cout` è normalmente connesso allo schermo.
k) La funzione membro `good()` restituisce `true` se le funzioni membro `bad()`, `fail()` e `eof()` restituiscono tutte `false`.
l) Lo stream `cin` è normalmente connesso allo schermo.
m) Se si verifica un errore irrecuperabile durante un'operazione su stream, la funzione membro `bad` restituisce `true`.
n) L'output su `cerr` non è bufferizzato, mentre l'output su `clog` è bufferizzato.
o) Quando è attivo il flag `ios::showpoint`, i valori a virgola mobile sono visualizzati con la precisione di default di sei cifre o, se il valore della precisione è stato modificato, con la precisione specificata.
p) La funzione membro `put` di `ostream` invia in output il numero specificato di caratteri.
q) I manipolatori di stream `dec`, `oct` e `hex` si applicano soltanto all'operazione di output immediatamente successiva.
r) In output gli indirizzi di memoria sono visualizzati per default come interi `long`.

11.3 Per ogni punto dell'esercizio scrivete una sola istruzione che compia l'operazione indicata.

a) Inviare in output la stringa "Enter your name: ".
b) Attivare il flag per visualizzare l'esponente in notazione scientifica e le lettere dei valori esadecimali in maiuscolo.
c) Inviare in output l'indirizzo di una variabile stringa di tipo `char *`.
d) Attivare il flag per visualizzare i valori a virgola mobile in notazione scientifica.
e) Inviare in output l'indirizzo della variabile `integerPtr` di tipo `int *`
f) Attivare un flag, in modo tale che nell'output dei valori ottali ed esadecimali sia visualizzata la loro base.
g) Inviare in output il valore a cui punta `floatPtr` di tipo `float *`
h) Utilizzare una funzione membro per impostare a `'*'` il carattere di riempimento dei campi più ampi dei valori inviati in output. *Scrivete un'altra istruzione che effettua lo stesso* compito con un manipolatore di stream.
i) Inviare in output i caratteri `'O'` e `'K'` in una sola istruzione con la funzione `put`.
j) Leggere il carattere successivo dallo stream di input senza estrarlo dallo stream.
k) Effettuare l'input e scartare i prossimi sei caratteri dallo stream di input standard.

l) Effettuare l'input di un carattere e memorizzarlo nella variabile `c` di tipo `char` tramite la funzione membro di `istream` `get` in due modi diversi.
m) Utilizzare la funzione membro di `istream` `read` per l'input di 50 caratteri da conservare nell'array `line` di tipo `char`.
n) Leggere 10 caratteri e conservarli nell'array di caratteri `name`. La lettura deve terminare quando si incontra il delimitatore `'.'`. Non eliminare il delimitatore dallo stream input. Scrivere un'altra istruzione che effettua lo stesso compito ed elimina il delimitatore dall'input.
o) Utilizzare la funzione membro di `istream` `gcount` per determinare il numero di caratteri ricevuti in input e memorizzati nell'array di caratteri `line` dall'ultima chiamata a `read` (funzione membro di `istream`) e il numero di caratteri inviati in output da `write` (funzione membro di `ostream`).
p) Scrivete istruzioni separate per svuotare (flush) lo stream di output utilizzando una funzione membro e un manipolatore di stream.
q) Inviare in output i seguenti valori: `124`, `18.376`, `'Z'`, `1000000` e `"String"`.
r) Visualizzare la precisione corrente tramite una funzione membro.
s) Effettuare l'input di un intero e memorizzarlo nella variabile `months` di tipo `int`, e l'input di un valore a virgola mobile e memorizzarlo nella variabile `percentageRate` di tipo `float`.
t) Visualizzare `1.92`, `1.925` e `1.9258` con 3 cifre di precisione tramite un manipolatore di stream.
u) Visualizzare l'intero `100` in formato ottale, esadecimale e decimale tramite un manipolatore di stream.
v) Visualizzare l'intero `100` in formato decimale, ottale ed esadecimale tramite un solo manipolatore di stream per modificare la base.
w) Visualizzare `1234` giustificato a destra in un campo di 10 cifre.
x) Leggere dei caratteri e memorizzarli nell'array `line` finché non si incontra il carattere `'z'` e fino a un massimo di 20 caratteri (incluso il carattere nullo finale). Non estrarre il delimitatore dallo stream.
y) Utilizzate le variabili intere `x` e `y` per specificare l'ampiezza del campo e la precisione che si devono utilizzare per visualizzare il valore `double` `87.4573`, e visualizzare tale valore.

11.4 Identificate l'errore in ognuna delle seguenti istruzioni e spiegate come correggerlo.

a)
```
cout << "Value of x <= y is: " << x <= y;
```
b) L'istruzione seguente deve visualizzare il valore intero di `'c'`
```
cout << 'c';
```
c)
```
cout << ""A string in quotes"";
```

11.5 Scrivete l'output prodotto dalle seguenti istruzioni.

a)
```
cout << "12345" << endl;
cout.width( 5 );
cout.fill( '*' );
cout << 123 << endl << 123;
```
b)
```
cout << setw( 10 ) << setfill( '$' ) << 10000;
```
c)
```
cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;
```
d)
```
cout << setiosflags( ios::showbase ) << oct << 99
     << endl << hex << 99;
```
e)
```
cout << 100000 << endl
     << setiosflags( ios::showpos ) << 100000;
```
f)
```
cout << setw( 10 ) << setprecision( 2 ) <<
   << setiosflags( ios::scientific ) << 444.93738;
```


Risposte agli esercizi di autovalutazione

11.1 a) `friend` b) `ios::left`, `ios::right` e `ios::internal`. c) stream o flussi. d) setiosflags, resetiosflags. e) iostream.h. f) iomanip.h. g) fstream.h. h) write. i) istream. lm) cerr o clog. m) ostream. n) <<. o) cin, cout, cerr e clog. p) >>. q) oct, hex, dec. r) sei cifre. s) ios::showpos.

11.2 a) Vero.
b) Falso. Gli operatori di inserimento nello stream e di estrazione dallo stream non sono in overloading per tutti i tipi definiti dall'utente. Chi crea una classe deve specificatamente fornire l'overloading dei due operatori per l'I/O del nuovo tipo di dato.
c) Falso. La funzione membro `flags()` senza argomenti restituisce semplicemente il valore corrente della variabile di stato flags.
d) Vero.
e) Vero.
f) Falso. Per effettuare l'overloading dell'operatore di inserimento nello stream <<, la funzione operatore deve prendere come argomenti un riferimento a `ostream` e un riferimento a un tipo definito dall'utente e deve restituire un riferimento a `ostream`.
g) Vero. A meno che `ios::skipws` sia inattivo.
h) Falso. Le funzionalità di I/O del C++ sono fornite come parte della libreria standard del C++. Il linguaggio C++ non prevede funzionalità per l'input, l'output o l'elaborazione dei file.
i) Vero.
j) Vero.
k) Vero.
l) Falso. Lo stream `cin` è connesso all'unità di input standard del computer, che coincide normalmente con la tastiera.
m) Vero.
n) Vero.
o) Vero.
p) Falso. La funzione membro di `ostream` `put` invia in output il suo argomento, cioè un carattere singolo.
q) Falso. I manipolatori di stream `dec`, `oct` e `hex` impostano la base degli interi finché questa non viene modificata nuovamente o fino al termine del programma.
r) Falso. Gli indirizzi di memoria sono visualizzati per default in formato esadecimale. Per visualizzarli come interi `long`, bisogna effettuarne il cast nel tipo `long`.

11.3 a)
```
cout << "Enter your name: ";
```
b)
```
cout.setf(ios::uppercase);
```
c)
```
cout << (void *) string;
```
d)
```
cout.setf(ios::scientific, ios::floatfield);
```
e)
```
cout << integerPtr;
```
f)
```
cout << setiosflags(ios::showbase);
```
g)
```
cout << *floatPtr;
```
h)
```
cout.fill( '*' );
cout << setfill( '*' );
```
i)
```
cout.put( 'O' ).put( 'K' );
```
j)
```
cin.peek();
```
k)
```
cin.ignore( 6 );
```
l)
```
c = cin.get();
cin.get( c );
```

```
m) cin.read( line, 50 );
n) cin.get( name, 10, '.' );
   cin.getline( name, 10, '.' );
o) cout.write( line, cin.gcount() );
p) cout.flush();
   cout << flush;
q) cout << 124 << 18.376 << 'Z' << 1000000 << "String";
r) cout << cout.precision();
s) cin >> months >> percentageRate;
t) cout << setprecision( 3 ) << 1.92 << '\t'
        << 1.925 << '\t' << 1.9258;
u) cout << oct << 100 << hex << 100 << dec << 100;
v) cout << 100 << setbase( 8 ) << 100 << setbase( 16 ) << 100;
w) cout << setw( 10 ) << 1234;
x) cin.get( line, 20, 'z' );
y) cout << setw( x ) << setprecision( y ) << 87.4573;
```

11.4 a) Errore: *la precedenza dell'operatore* `<<` *è più alta di quella dell'operatore* `<=`, *per cui* l'istruzione verrà valutata in modo scorretto e causerà un messaggio di errore del compilatore.
Correzione: Per correggere l'istruzione, racchiudete tra parentesi l'espressione `x <= y`. Questo problema si presenterà con ogni espressione che utilizza operatori con precedenza più bassa di `<<`, *se l'espressione non è racchiusa tra parentesi.*

b) Errore: In C++ l'operatore << sui caratteri non li tratta come valori interi, come invece accade in C.
Correzione: Per visualizzare il valore numerico di un carattere secondo il set di caratteri del proprio computer, occorre effettuarne il cast verso il tipo `int` come segue:
`cout << (int)'c';`

c) Errore: I doppi apici non possono essere visualizzati in una stringa, a meno che non si utilizzi la sequenza di escape \".
Correzione: Visualizzate la stringa in uno dei due modi seguenti:
`cout << '"' << "A string in quotes" << '"';`
`cout << "\"A string in quotes\"";`

11.5 a)
```
12345
**123
123
```
b)
```
SSSSS10000
```
c)
```
1024.988
```
d)
```
0143
0x63
```
e)
```
100000
+100000
```
f)
```
4.45e+02
```


Esercizi

11.6 Scrivete un'istruzione per ognuno dei seguenti compiti:
a) Visualizzare l'intero `40000` giustificato a sinistra in un campo di 15 cifre.
b) Leggere una stringa e memorizzarla nell'array di caratteri `state`.
c) Visualizzare 200 con e senza il suo segno.
d) Visualizzare il valore decimale `100` in formato esadecimale, con il prefisso `0x`.
e) Leggere dei caratteri nell'array `s` finché non si incontra il carattere `'p'` o non si raggiunge il limite di 10 caratteri (incluso il carattere nullo finale). Estrarre il delimitatore dallo stream di input e scartarlo.
f) Visualizzare `1.234` in un campo di 9 cifre preceduto da cifre zero.
g) Leggere una stringa della forma `"characters"` dall'input standard. Memorizzare la stringa nell'array di caratteri `s`. Eliminare gli apici dallo stream di input. Leggere un massimo di 50 caratteri (incluso il carattere nullo finale).

11.7 Scrivete un programma per l'input di valori interi nei formati decimale, ottale ed esadecimale. Inviate in output ogni intero letto dal programma in tutti e tre i formati. Verificate il funzionamento del programma con i seguenti valori: `10`, `010`, `0x10`.

11.8 Scrivete un programma che visualizza i valori dei puntatori utilizzando il cast verso un tipo di dato intero. Quali valori sono visualizzati in modo bizzarro? Quali valori causano degli errori?

11.9 Scrivete un programma che visualizza il valore intero `12345` e il valore a virgola mobile `1.2345` in campi di diverse dimensioni. Che cosa succede quando si visualizzano i valori in campi di dimensioni inferiore alle cifre che contengono?

11.10 Scrivete un programma che visualizza il valore `100.453627` arrotondandolo all'intero, ai decimi, ai centesimi e ai millesimi più vicini.

11.11 Scrivete un programma che effettua l'input di una stringa dalla tastiera e ne determina la lunghezza. Visualizzate la stringa in un campo ampio il doppio della lunghezza della stringa.

11.12 Scrivete un programma che converte le temperature intere espresse in gradi Fahrenheit (0-212) in valori a virgola mobile Celsius con 3 cifre di precisione. Utilizzate la formula

```
celsius = 5.0 / 9.0 * ( fahrenheit - 32 );
```

per il calcolo. L'output deve essere visualizzato in due colonne giustificate a destra, e i gradi Celsius devono avere sempre il segno, sia esso negativo o positivo.

11.13 In alcuni linguaggi di programmazione, le stringhe sono immesse racchiudendole tra apici *singoli o doppi. Scrivete un programma che legge le tre stringhe* `suzy`, `"suzy"` *e* `'suzy'` *Gli apici sono* letti come parti della stringa o ignorati?

11.14 Nella Figura 8.3, gli operatori di inserimento/estrazione hanno subito overloading per l'I/O degli oggetti della classe `PhoneNumber`. Riscrivete l'operatore di estrazione dallo stream per effettuare i seguenti controlli sull'input.
a) Effettuate l'input di un numero telefonico in un array verificando che siano stati digitati 14 caratteri nella forma `(800) 555-1212`.
b) Verificate che il prefisso e il primo gruppo di cifre non inizi per `0` o `1`.
c) Verificate che la cifra centrale del prefisso sia `0` o `1`.

In tutti e tre i casi utilizzate la funzione membro `clear` per attivare il flag `ios::failbit` in caso di input scorretto. Se nessuna di queste operazioni ha attivato `ios::failbit`, copiate le tre parti del numero nei membri `areaCode`, `exchange` e `line` dell'oggetto `PhoneNumber`. Nel programma di prova, se si riscontra l'attivazione di `ios::failbit`, prevedete la visualizzazione di un messaggio di errore e la terminazione immediata del programma.

11.15 Scrivete un programma secondo le seguenti indicazioni:

a) Create la classe `Point` contenente i dati membri `private` interi `xCoordinate` e `yCoordinate`, e dichiara l'overloading delle funzioni di estrazione/inserimento per l'I/O su stream come `friend`.

b) Definite le funzioni operatore di estrazione/inserimento. Esse dovrebbero contenere dei controlli per verificare la validità dei dati, e in caso di errore dovrebbero attivare `ios::failbit`. L'operatore di inserimento nello stream non dovrebbe visualizzare il punto se si è verificato un errore.

Scrivete un programma di prova per l'I/O del tipo di dato `Point` tramite gli operatori di estrazione/inserimento.

11.16 Scrivete un programma secondo le seguenti indicazioni:

a) Create la classe `Complex` (per i numeri complessi) contenente i dati membro `private` interi `real` e `imaginary`, e che dichiara gli overloading degli operatori di inserimento/estrazione come `friend`.

b) Definite le funzioni operatore di inserimento/estrazione. Esse dovrebbero contenere dei controlli per verificare la validità dei dati, e in caso di errore dovrebbero attivare `ios::failbit`. L'input dovrebbe essere della forma:
`3 + 8i`

c) I valori possono essere negativi o positivi ed è possibile non fornire uno dei due valori, nel qual caso il dato membro corrispondente deve essere impostato a `0`.

L'operatore di inserimento nello stream non dovrebbe visualizzare il numero complesso se si è verificato un errore. Il formato dell'output deve essere identico al formato dell'input con la convenzione che, per i valori immaginari negativi, deve essere visualizzato un segno meno e non un segno più.
Scrivete un programma di prova per l'I/O del tipo di dato `Complex` tramite gli operatori di estrazione/inserimento.

11.17 Scrivete un programma che contiene un ciclo `for` per visualizzare una tabella dei caratteri che vanno dal valore ASCII 33 al valore 126. Il programma deve visualizzare il valore decimale, ottale, esadecimale e il valore carattere di ciascuno. Utilizzate i manipolatori di stream `dec`, `oct` e `hex` per visualizzare i valori interi.

11.18 Scrivete un programma per dimostrare che sia la `getline` che la `get` a tre argomenti terminano l'input di una stringa con un carattere nullo. `get` inoltre lascia il delimitatore nello stream di input mentre `getline` lo estrae e lo scarta. Che cosa succede ai caratteri non letti nello stream?

11.19 Scrivete un programma che crea il manipolatore definito dall'utente `skipwhite` per saltare i caratteri di spaziatura iniziali in uno stream di input. Il manipolatore deve utilizzare la funzione `isspace` della libreria di `ctype.h` per verificare se il carattere è un carattere di spaziatura. Ogni carattere deve essere ottenuto dall'input tramite la funzione `get`. Quando si incontra un carattere non di spaziatura, `skipwhite` termina il suo compito rimettendolo nello stream di input e restituendo un riferimento a `istream`.
Per verificare il funzionamento del vostro manipolatore scrivete un programma di prova in cui il flag `ios::skipws` viene disattivato, in modo che l'operatore di estrazione dallo stream non salti automaticamente i caratteri di spaziatura. Quindi verificate il manipolatore immettendo nell'input un carattere preceduto da un carattere di spaziatura. Visualizzate il carattere ricevuto in input, per confermare che il carattere di spaziatura è stato scartato.

APPENDICE A

Riepilogo degli operatori

Elenchiamo di seguito gli operatori in ordine decrescente di precedenza.

Operatore	Tipo	Associatività
`::`	risoluzione dello scope (binario)	da sx a dx
`::`	risoluzione dello scope (unario)	da sx a dx
`()`	parentesi	da sx a dx
`[]`	indicizzazione di array	da sx a dx
`.`	selezione del membro tramite un oggetto	da sx a dx
`->`	selezione del membro tramite un puntatore	da sx a dx
`++`	postincremento (unario)	da dx a sx
`--`	postdecremento (unario)	da dx a sx
`typeid`	informazioni di tipo a run-time	da dx a sx
`dynamic_cast< type >`	cast con controllo di tipo a run-time	da dx a sx
`static_cast< type >`	cast con controllo di tipo durante la compilazione	da dx a sx
`reinterpret_cast< type >`	cast per conversioni non standard	da dx a sx
`const_cast< type >`	eliminazione del carattere `const`	da dx a sx
`++`	postincremento (unario)	da dx a sx
`--`	postdecremento (unario)	da dx a sx
`+`	più (unario)	da dx a sx
`-`	meno (unario)	da dx a sx
`!`	negazione logica (unario)	da dx a sx
`~`	complemento a bit (unario)	da dx a sx
`( type )`	cast unario in stile C	da dx a sx
`sizeof`	memoria occupata in byte	da dx a sx
`&`	indirizzo	da dx a sx
`*`	indirezione	da dx a sx
`new`	allocazione dinamica della memoria	da dx a sx
`new[]`	allocazione dinamica di array	da dx a sx
`delete`	deallocazione dinamica della memoria	da dx a sx
`delete[]`	deallocazione dinamica di array	da dx a sx
`.*`	puntatore al membro tramite un oggetto	da sx a dx

Figura A.1 Schema di riepilogo degli operatori (continua)

Operatore	Tipo	Associatività
->*	puntatore al membro tramite un puntatore	da sx a dx
*	moltiplicazione	da sx a dx
/	divisione	da sx a dx
%	modulo	da sx a dx
+	addizione	da sx a dx
-	sottrazione	da sx a dx
<<	shifting a sx dei bit	da sx a dx
>>	shifting a dx dei bit	da sx a dx
<	relazionale, minore di	da sx a dx
<=	relazionale, minore di o uguale a	da sx a dx
>	relazionale, maggiore di	da sx a dx
>=	relazionale, maggiore di o uguale a	da sx a dx
==	relazionale, uguale a	da sx a dx
!=	relazionale, diverso da	da sx a dx
&	AND su bit	da sx a dx
^	OR esclusivo su bit	da sx a dx
|	OR inclusivo su bit	da sx a dx
&&	AND logico	da sx a dx
||	OR logico	da sx a dx
?:	condizionale (ternario)	da dx a sx
=	assegnamento	da dx a sx
+=	assegnamento con addizione	da dx a sx
-=	assegnamento con sottrazione	da dx a sx
*=	assegnamento con moltiplicazione	da dx a sx
/=	assegnamento con divisione	da dx a sx
%=	assegnamento con modulo	da dx a sx
&=	assegnamento con AND su bit	da dx a sx
^=	assegnamento con OR esclusivo su bit	da dx a sx
|=	assegnamento con OR inclusivo su bit	da dx a sx
<<=	assegnamento con shifting a sx su bit	da dx a sx
>>=	assegnamento con shifting a dx su bit	da dx a sx
,	virgola	da sx a dx

Figura A.1 Schema di riepilogo degli operatori.

APPENDICE B

L'insieme dei caratteri ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:;	
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	lm	
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

I numeri a sinistra della tabella rappresentano le cifre più significative del codice del carattere, espresso in notazione decimale (0-127), mentre quelli in cima alla tabella rappresentano le cifre meno significative del codice del carattere. Per esempio, il codice del carattere 'F' è 70, mentre quello di '&' è 38.

Nota: L'insieme di caratteri ASCII è un sottogruppo dell'insieme di caratteri Unicode utilizzati da Java per rappresentare i caratteri delle principali lingue del mondo. Per ulteriori informazioni circa l'insieme di caratteri Unicode, è possibile visitare il sito `http://unicode.org`.

APPENDICE C

I sistemi di numerazione

Obiettivi

- Apprendere i concetti fondamentali riguardanti i sistemi di numerazione
- Capire come operare con i numeri rappresentati nei sistemi di numerazione binario, ottale ed esadecimale
- Imparare ad usare la notazione ottale o esadecimale per abbreviare i numeri binari
- Imparare a convertire i numeri ottali ed esadecimali in quelli binari
- Imparare a convertire i numeri decimali nei loro equivalenti binari, ottali ed esadecimali e viceversa
- Apprendere l'aritmetica binaria e il modo in cui i numeri binari sono rappresentati utilizzando la notazione in complemento a due

C.1 Introduzione

Questa appendice prenderà in esame i principali sistemi di numerazione utilizzati dai programmatori C++, specialmente da chi si trova a lavorare su progetti software che richiedono una stretta interazione con i componenti hardware quali, ad esempio: i sistemi operativi, il software di rete, i compilatori, i sistemi di basi di dati e le applicazioni che richiedano prestazioni elevate.

Quando, all'interno di un programma in C++, scrivete un numero intero come 227 o –63, questo è interpretato come appartenente al *sistema numerico decimale (base 10)*. Le *cifre* del sistema numerico decimale sono 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9, dove 0 è la cifra più bassa mentre 9 è quella più alta (uno in meno della base 10). Internamente, però, i computer utilizzano il *sistema numerico binario (base 2)*, che è costituito soltanto da due cifre, ovverosia 0 e 1; in questo caso, 0 è la cifra più bassa, mentre 1 è quella più alta (uno in meno della base 2).

Come vedrete, i numeri binari tendono a essere più lunghi dei loro equivalenti decimali. I programmatori che lavorano con i linguaggi assembly e con quelli di alto livello (come il C), che permettono loro di arrivare fino al "livello macchina", trovano normalmente molto scomodi i numeri binari; è questo il motivo per cui sono diventati popolari due altri sistemi di numerazione, ovverosia il *sistema numerico ottale (base 8)* e il *sistema numerico esadecimale (base 16)*, che costituiscono un'abbreviazione dei numeri binari.

Nel sistema numerico ottale, le cifre vanno dallo 0 al 7; giacché sia il sistema numerico binario sia quello ottale utilizzano meno cifre di quello decimale, queste sono uguali alle loro corrispondenti decimali.

Il sistema numerico esadecimale presenta un problema, giacché richiede sedici cifre (0, quella più bassa, e una più alta con un valore equivalente al 15 decimale, ovverosia uno in meno della base 16). La convenzione vuole che siano utilizzate le lettere dalla A alla F per rappresentare le cifre esadecimali corrispondenti ai valori decimali dal 10 al 15; nel sistema esadecimale, quindi, potranno esservi dei numeri come 876 consistenti solamente di cifre simili a quelle decimali, oppure dei numeri come 8A55F consistenti di cifre e lettere, oppure ancora numeri come FFE consistenti soltanto di lettere. A volte, un numero esadecimale potrà assomigliare a una parola come FEDE o CAFFE; tutto ciò potrà sembrare strano ai programmatori abituati a lavorare con i numeri.

Ognuno di questi sistemi di numerazione utilizza la *notazione posizionale*, ovverosia *ogni posizione in cui è stata scritta una cifra possiede un diverso valore posizionale*. Nel caso del numero decimale 937 (il 9, il 3 e il 7 sono chiamati *valori simbolo*), per esempio, il 7 è scritto nella *posizione delle unità*, il 3 in quella delle *decine* e il 9 in quella delle *centinaia*. Osservate che ognuna di queste posizioni è una potenza della base (10) e che queste potenze partono da 0 e crescono di 1 man mano che ci si sposta a sinistra all'interno del numero (Fig. C.3).

Cifra binaria	Cifra ottale	Cifra decimale	Cifra esadecimale
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valore decimale di10)
			B (valore decimale di 11)
			C (valore decimale di 12)
			D (valore decimale di 13)
			E (valore decimale di 14)
			F (valore decimale di 15)

Figura C.1 Le cifre dei sistemi di numerazione binario, ottale, decimale ed esadecimale.

Attributo	Binario	Ottale	Decimale	Esadecimale
Base	2	8	10	16
Cifra più bassa	0	0	0	0
Cifra più alta	1	7	9	F

Figura C.2 Confronto tra i sistemi di numerazione binario, ottale, decimale ed esadecimale.

I valori posizionali nel sistema numerico decimale			
Cifra decimale	9	3	7
Nome della posizione	Centinaia	Decine	Unità
Valore posizionale	100	10	1
Valore posizionale espresso come potenza della base (10)	10^2	10^1	10^0

Figura C.3 I valori posizionali nel sistema numerico decimale.

Nel caso dei numeri decimali più lunghi, le posizioni successive a sinistra corrisponderebbero alle *migliaia* (10 alla terza potenza), alle *decine di migliaia* (10 alla quarta), alle *centinaia di migliaia* (10 alla quinta), ai *milioni* (10 alla sesta), alle *decine di milioni* (10 alla settima), e così via.

Nel numero binario 101, si dice che l'uno all'estrema destra si trova nella *posizione degli uno*, lo 0 in quella *dei due* e l'uno all'estrema sinistra in quella *dei quattro*. Notate che ognuna di queste posizioni è una potenza della base (2), e che queste potenze partono da 0 e aumentano di 1 man mano che ci si sposta a sinistra all'interno del numero (Fig. C.4).

Nel caso dei numeri binari più lunghi, le posizioni successive a sinistra corrisponderebbero a quella degli *otto* (2 alla terza potenza), a quella dei *sedici* (2 alla quarta), a quella dei *trentadue* (2 alla quinta), a quella dei *sessantaquattro* (2 alla sesta), e così via.

Nel numero ottale 425, si dice che il 5 si trova nella *posizione degli uno*, il due in quella *degli otto* e il 4 in quella *dei sessantaquattro*. Notate che ognuna di queste posizioni è una potenza della base (8), e che queste potenze partono da 0 e aumentano di 1 man mano che ci si sposta a sinistra all'interno del numero (Fig. C.5).

Nel caso dei numeri ottali più lunghi, le posizioni successive a sinistra corrisponderebbero a quella dei *cinquecentododici* (8 alla terza potenza), a quella dei *quattromilanovantasei* (8 alla quarta), a quella dei *trentaduemilasettecentosessantotto* (8 alla quinta), e così via.

Nel numero esadecimale 3DA, si dice che la A si trova nella *posizione degli uno*, la D in quella *dei sedici* e il 3 in quella *dei duecentocinquantasei*. Notate che ognuna di queste posizioni è una potenza della base (16), e che queste potenze partono da 0 ed aumentano di 1 man mano che ci si sposta a sinistra all'interno del numero (Fig. C.6).

I valori posizionali nel sistema numerico binario			
Cifra binaria	1	0	1
Nome della posizione	Quattro	Due	Uno
Valore posizionale	4	2	1
Valore posizionale espresso come potenza della base (2)	2^2	2^1	2^0

Figura C.4 I valori posizionali nel sistema numerico binario.

I valori posizionali nel sistema numerico ottale			
Cifra decimale	4	2	5
Nome della posizione	Sessantaquattro	Otto	Uno
Valore posizionale	64	8	1
Valore posizionale espresso come potenza della base (8)	8^2	8^1	8^0

Figura C.5 I valori posizionali nel sistema numerico ottale.

I valori posizionali nel sistema numerico esadecimale			
Cifra decimale	3	D	A
Nome della posizione	Duecentocinquantasei	Sedici	Uno
Valore posizionale	256	16	1
Valore posizionale espresso come potenza della base (16)	16^2	16^1	16^0

Figura C.6 I valori posizionali nel sistema numerico esadecimale.

Nel caso dei numeri esadecimali più lunghi, le posizioni successive a sinistra corrisponderebbero a quella *dei quattromilanovantasei* (16 alla terza potenza), a quella dei *sessantacinquemilacinquecentotrentasei* (16 alla quarta), e così via.

C.2 Usare i numeri ottali ed esadecimali per abbreviare i numeri binari

Il principale impiego dei numeri ottali ed esadecimali in ambito informatico è l'abbreviazione delle rappresentazioni binarie più lunghe. La Fig. C.7 mostra come i numeri binari possono essere espressi in modo più conciso all'interno dei sistemi di numerazione con basi più alte rispetto a quello binario.

Numero decimale	Rappresentazione binaria	Rappresentazione ottale	Rappresentazione esadecimale
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A

Figura C.7 Gli equivalenti decimali, binari, ottali ed esadecimali (continua)

Numero decimale	Rappresentazione binaria	Rappresentazione ottale	Rappresentazione esadecimale
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Figura C.7 Gli equivalenti decimali, binari, ottali ed esadecimali.

Il sistema numerico ottale e quello esadecimale sono in una relazione particolarmente importante con quello binario: le basi degli ottali e degli esadecimali (rispettivamente 8 e 16), infatti, sono potenze di quella del sistema numerico binario (base 2). Per esempio, prendete in considerazione il seguente numero binario a 12 cifre con i suoi equivalenti ottale ed esadecimale, e cercate di determinare il modo in cui questa relazione facilita l'abbreviazione dei numeri binari in ottali o esadecimali (la risposta si trova dopo i numeri).

Numero binario	Equivalente ottale	Equivalente esadecimale
100011010001	4321	8D1

Per vedere in che modo un numero binario possa essere facilmente convertito in un ottale, sarà sufficiente suddividere le 12 cifre del numero binario in gruppi di tre bit consecutivi, scrivendo poi questi gruppi al posto delle corrispondenti cifre del numero ottale, così:

```
100     011     010     001
4       3       2       1
```

Notate che la cifra ottale scritta sotto ogni gruppo di tre bit corrisponde precisamente all'equivalente ottale di quel numero binario di 3 cifre, come mostrato nella Fig. C.7.

Lo stesso tipo di relazione si potrà osservare nella conversione dei numeri da binari a esadecimali. In particolare, potete provare a suddividere le 12 cifre del numero binario in gruppi di quattro bit consecutivi, scrivendo poi questi gruppi al posto delle corrispondenti cifre del numero esadecimale, così:

```
1000    1101    0001
8       D       1
```

Notate che la cifra esadecimale scritta sotto ogni gruppo di quattro bit corrisponde precisamente all'equivalente esadecimale di quel numero binario di 4 cifre, come mostrato nella Fig. C.7.

C.3 La conversione dei numeri ottali ed esadecimali in binari

Nella sezione precedente avete visto come sia possibile convertire i numeri binari nei loro equivalenti ottali ed esadecimali, attraverso la creazione di gruppi di cifre binarie e la loro successiva scrittura nei valori ottali o esadecimali equivalenti. Questo processo potrà essere utilizzato anche in senso inverso, al fine di produrre l'equivalente binario di un dato numero ottale o esadecimale.

Il numero ottale 653, per esempio, sarà convertito in binario semplicemente scrivendo il 6 nel suo equivalente binario di tre cifre 110, il 5 nel suo equivalente binario 101 e il 3 nel suo equivalente binario 011, ottenendo così il numero binario di nove cifre 110101011.

Il numero esadecimale FAD5, invece, sarà convertito in binario semplicemente scrivendo la F nel suo equivalente binario di 4 cifre 1111, la A nel suo equivalente binario 1010, la D nel suo equivalente binario 1101 e il 5 nel suo equivalente binario 0101, ottenendo così il numero 1111101011010101, formato da 16 cifre.

C.4 La conversione da binario, ottale o esadecimale in decimale

Dato che siamo tutti normalmente abituati a utilizzare i numeri decimali, spesso è più comodo convertire un numero binario, ottale o esadecimale in decimale, al fine di capire meglio quale sia il valore del numero in questione nel sistema di numerazione che usiamo più comunemente. I diagrammi della Sezione C.1 esprimono i valori posizionali in notazione decimale; per convertire un numero in decimale sarà necessario moltiplicare l'equivalente decimale di ogni cifra per il suo valore posizionale e sommare questi prodotti. Il numero binario 110101, per esempio, sarà convertito nel decimale 53 nel modo mostrato dalla Fig. C.8.

Convertire un numero binario in decimale

Valori posizionali:	32	16	8	4	2	1
Valori simbolo:	1	1	0	1	0	1
Prodotti:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Somma:	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

Figura C.8 Convertire un numero binario in decimale.

Con questa stessa tecnica, per convertire l'ottale 7614 nel decimale 3980 sarà necessario utilizzare i valori posizionali ottali appropriati, come mostrato nella Fig. C.9.

Per finire, per convertire il valore esadecimale AD3B nel decimale 44347 sarà necessario utilizzare i valori posizionali esadecimali appropriati, come mostrato nella Fig. C.10.

Convertire un numero ottale in decimale

Valori posizionali	512	64	8	1
Valori simbolo:	7	6	1	4
Prodotti:	7*512=3584	6*64=384	1*8=8	4*1=4
Somma:	= 3584 + 384 + 8 + 4 = 3980			

Figura C.9 Convertire un numero ottale in decimale.

Convertire un numero esadecimale in decimale

Valori posizionali	4096	256	16	1
Valori simbolo:	A	D	3	B
Prodotti:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Somma:	= 40960 + 3328 + 48 + 11 = 44347			

Figura C.10 Convertire un numero esadecimale in decimale.

C.5 La conversione da decimale a binario, ottale o esadecimale

Le conversioni viste nelle sezioni precedenti seguono le convenzioni della notazione posizionale e lo stesso varrà anche per la conversione da decimale a binario, ottale o esadecimale.

Supponendo di voler convertire il valore decimale 57 in binario, sarà necessario partire dalla scrittura dei valori posizionali delle colonne da destra a sinistra, fino a che non sarà stata raggiunta una colonna il cui valore posizionale sia maggiore del numero decimale; questa colonna non sarà necessaria e quindi dovrà essere ignorata. In conformità a quanto appena affermato, ecco cosa bisognerà scrivere:

Valori posizionali: 64 32 16 8 4 2 1

ignorando la colonna con il valore posizionale 64 si otterrà:

Valori posizionali: 32 16 8 4 2 1

A questo punto, sarà necessario operare sulle colonne da sinistra verso destra. Dividendo 57 per 32, si otterrà un risultato di 1 con un resto di 25, quindi sarà necessario scrivere 1 nella colonna del 32. Dividendo 25 per 16, si otterrà un risultato di 1 con un resto di 9 e bisognerà quindi scrivere 1 nella colonna del 16. Dividendo 9 per 8, si otterrà un risultato di 1 con un resto di 1. Le due colonne successive produrranno ognuna quozienti uguali a zero quando i loro valori posizionali saranno divisi per 1, quindi sarà necessario scrivere 0 nelle colonne del 4 e del 2. Per finire, 1 diviso 1 farà 1 e quindi sarà necessario scrivere 1 nella colonna dell'uno. Ecco il risultato che si otterrà:

Valori posizionali: 32 16 8 4 2 1

Valori simbolo: 1 1 1 0 0 1

e quindi il valore decimale 57 è equivalente al valore binario 111001.

Per convertire il valore decimale 103 in ottale, bisognerà per prima cosa scrivere i valori posizionali delle colonne fino a che non sarà stata raggiunta una colonna il cui valore posizionale sia maggiore del numero decimale; questa colonna non sarà necessaria e quindi sarà ignorata. In conformità a quanto appena affermato, ecco cosa bisognerà scrivere:

Valori posizionali: 512 64 8 1

ignorando la colonna con il valore posizionale 512 si otterrà:

Valori posizionali: 64 8 1

A questo punto, sarà necessario operare sulle colonne da sinistra verso destra. Dividendo 103 per 64, si otterrà un risultato di 1 con un resto di 39 e quindi sarà necessario scrivere 1 nella colonna del 64. Dividendo 39 per 8, si otterrà un risultato di 4 con un resto di 7 e bisognerà quindi scrivere 4 nella colonna dell'otto. Dividendo 7 per 1, si otterrà un risultato di 7 con un resto di 0 e quindi bisognerà scrivere 7 nella colonna dell'uno. Ecco il risultato che si otterrà:

Valori posizionali: 64 8 1

Valori simbolo: 1 4 7

e quindi il valore decimale 103 è equivalente al valore ottale 147.

Per convertire il valore decimale 375 in esadecimale, bisognerà per prima cosa scrivere i valori posizionali delle colonne fino a che non sarà stata raggiunta una colonna il cui valore posizionale sia maggiore del numero decimale; questa colonna non sarà necessaria e quindi sarà ignorata. In conformità a quanto appena affermato, ecco cosa bisognerà scrivere:

Valori posizionali: 4096 256 16 1

ignorando la colonna con il valore posizionale 4096 si otterrà:

Valori posizionali: 256 16 1

A questo punto, sarà necessario operare sulle colonne da sinistra verso destra. Dividendo 375 per 256, si otterrà un risultato di 1 con un resto di 119 e quindi sarà necessario scrivere 1 nella colonna del 256. Dividendo 119 per 16, si otterrà un risultato di 7 con un resto di 7 e bisogna quindi scrivere 7 nella colonna del 16. Dividendo 7 per 1, si otterrà un risultato di 7 con un resto di 0 e quindi bisognerà scrivere 7 nella colonna dell'uno. Ecco il risultato che si otterrà:

Valori posizionali: 256 16 1

Valori simbolo: 1 7 7

e quindi il valore decimale 375 è equivalente al valore esadecimale 177.

C.6 I numeri binari negativi: la notazione in complemento a due

Finora questa appendice ha preso in esame soltanto i numeri positivi; i computer, però, sono in grado di rappresentare anche i numeri negativi, attraverso la *notazione in complemento a due*. Per prima cosa, spiegheremo come si forma il complemento a due di un numero binario, dopodiché passeremo ad illustrare in che modo esso rappresenti il valore negativo di un numero binario.

Pensate a una macchina con numeri interi a 32 bit, supponendo che:

```
int value = 13;
```

La rappresentazione a 32 bit di `value` sarà

```
00000000 00000000 00000000 00001101
```

Per formare il negativo di `value`, sarà necessario per prima cosa formare il suo *complemento a uno*, applicando l'operatore C di complemento bitwise (~):

```
ones_complement_of_value = ~value;
```

Internamente, `~value` corrisponderà ora a `value` con tutti i suoi bit invertiti (gli uno saranno diventati zeri e gli zeri saranno diventati uno):

```
value:
00000000 00000000 00000000 00001101

~value (ovvero il complemento a uno di value):
11111111 11111111 11111111 11110010
```

Per formare il complemento a due di `value`, basterà semplicemente aggiungere uno al complemento a uno di `value`:

```
Complemento a due di value:
11111111 11111111 11111111 11110011
```

Ora, se questo fosse effettivamente equivalente a –13, dovremmo essere in grado di aggiungerlo al binario 13 ottenendo un risultato pari a 0:

```
  00000000 00000000 00000000 00001101
+ 11111111 11111111 11111111 11110011
---------------------------------------
  00000000 00000000 00000000 00000000
```

Il riporto risultante dalla colonna all'estrema sinistra sarà ignorato e quindi si otterrà effettivamente zero come risultato. Aggiungendo il complemento a uno di un numero a quello stesso numero, il risultato sarebbe composto interamente da tanti 1. Il motivo per cui si otterrà un risultato composto solamente da zeri, invece, è che il complemento a due è 1 in più del complemento a uno; l'addizione di questo 1 farà in modo che ogni colonna sommi lo 0 con un riporto di 1, e questo riporto continuerà a essere trasportato a sinistra fino a quando non sarà ignorato nell'ultimo bit, producendo così un risultato di soli zeri.

I computer eseguono in realtà una sottrazione di questo tipo:

```
x = a - value;
```

aggiungendo ad `a` il complemento a due di `value` nel seguente modo:

```
x = a + (~value + 1);
```

Supponete ora che `a` sia 27 e che `value` sia 13; se il complemento a due di `value` fosse realmente il negativo di `value`, aggiungendo ad `a` il complemento a due di `value` si dovrebbe ottenere 14:

```
a (ovverosia 27)       00000000 00000000 00000000 00011011
+ (~value + 1)        +11111111 11111111 11111111 11110011
                       -----------------------------------
                       00000000 00000000 00000000 00001110
```

che equivale effettivamente a 14.

Esercizi di autovalutazione

C.1 Le basi dei sistemi di numerazione decimale, binario, ottale ed esadecimale sono rispettivamente _____, _____, _____ e _____.

C.2 In generale, le rappresentazioni decimali, ottali ed esadecimali di un dato numero binario contengono (più/meno) cifre rispetto a quelle contenute nel numero binario?

C.3 (Vero/Falso) Uno dei motivi per cui è utilizzato il sistema numerico decimale è che permette di abbreviare i numeri binari semplicemente sostituendo una cifra decimale per ogni gruppo di quattro bit binari.

C.4 La rappresentazione (ottale / esadecimale / decimale) di un grande valore binario è la più concisa (tra le alternative proposte).

C.5 (Vero / Falso) La cifra più alta in qualsiasi base è una in più della base.

C.6 (Vero / Falso) La cifra più bassa in qualsiasi base è una in meno della base.

C.7 Il valore posizionale della cifra all'estrema destra di qualsiasi numero binario, ottale, decimale o esadecimale è sempre _____.

C.8 Il valore posizionale della cifra che si trova alla sinistra della cifra all'estrema destra di un qualsiasi numero binario, ottale, decimale o esadecimale è sempre pari a _____.

C.9 Completate questa tabella con i valori posizionali relativi alle quattro posizioni all'estrema destra di ognuno dei sistemi di numerazione indicati:

decimale	1000	100	10	1
esadecimale	_	256	_	_
binario	_	_	_	_
ottale	512	_	8	_

C.10 Convertite il valore binario `110101011000` in ottale e in esadecimale.

C.11 Convertite il valore esadecimale `FACE` in binario.

C.12 Convertite il valore ottale `7316` in binario.

C.13 Convertite il valore esadecimale `4FEC` in ottale. (Suggerimento: convertite prima `4FEC` in binario e poi convertite questo valore binario in ottale).

C.14 Convertite il valore binario `1101110` in decimale.

C.15 Convertite il valore ottale `317` in decimale.

C.16 Convertite il valore esadecimale `EFD4` in decimale.

C.17 Convertite il valore decimale `177` in binario, ottale ed esadecimale.

C.18 Mostrate la rappresentazione binaria del decimale `417`, quindi mostrate il complemento a uno di `417` e il complemento a due di `417`.

C.19 Quale risultato si otterrà aggiungendo il complemento a uno di un numero a quel numero?

Risposte agli esercizi di autovalutazione

C.1 `10; 2; 8; 16.`

C.2 Meno.

C.3 Falso.

C.4 Esadecimale.

C.5 Falso. La cifra più alta in qualunque base è uno meno della base.

C.6 Falso. La cifra più bassa di qualunque base è zero.

C.7 1 (la base elevata alla potenza zero).

C.8. La base del sistema numerico.

C.9 Completate questa tabella con i valori posizionali relativi alle quattro posizioni all'estrema destra di ognuno dei sistemi di numerazione indicati:

decimale	1000	100	10	1
esadecimale	4096	256	16	1
binario	8	4	2	1
ottale	512	64	8	1

C.10 Ottale `6530`; Esadecimale `D58`.

C.11 Binario `1111 1010 1100 1110`.

C.12 Binario `111 011 001 110`.

C.13 Binario `0 100 111 111 101 100`; Ottale `47754`.

C.14 Decimale `2+4+8+32+64=110`.

C.15 Decimale `7+1*8+3*64=7+8+192=207`.

C.16 Decimale `4+13*16+15*256+14*4096=61396`.

C.17 Decimale `177`

in binario:

```
256    128    64     32     16     8      4      2      1
128    64     32     16     8      4      2      1
(1*128) + (0*64) + (1*32) + (1*16) + (0*8) + (0*4) + (0*2) + (1*1)
10110001
```

in ottale:

```
512    64     8      1
64     8      1
(2*64) + (6*8) + (1*1)
261
```

in esadecimale:

```
256    16     1
16     1
(11*16) + (1*1)
(B*16) + (1*1)
B1
```

C.18 *Binario:*

```
512   256   128   64    32    16    8     4     2     1
256   128   64    32    16    8     4     2     1
(1*256) + (1*128) + (0*64) + (1*32) + (0*16) + (0*8) + (0*4) + (0*2) + (1*1)
110100001
```

Complemento a uno: `001011110`
Complemento a due: `001011111`
Controllo: il numero binario originale + il suo complemento a due

```
110100001
001011111
---------
000000000
```

C.19 Zero.

Esercizi

C.20 Alcune persone sostengono che molti dei nostri calcoli sarebbero più semplici con il sistema *numerico in base 12, poiché il 12 è divisibile* per molti più numeri rispetto al 10 (del sistema in base 10). Qual è la cifra più bassa della base 12? Quale potrebbe essere il simbolo per la cifra più alta in base 12? Quali sono i valori posizionali delle quattro posizioni all'estrema destra di qualsiasi numero del sistema numerico in base 12?

C.21 *Nei sistemi* di numerazione presi in esame in questo capitolo, qual è il rapporto tra il valore del simbolo più alto e quello posizionale della prima cifra che si trova alla sinistra di quella all'estrema destra di ogni numero?

C.22 Completate questa tabella con i valori posizionali relativi alle quattro posizioni all'estrema destra di ognuno dei sistemi di numerazione indicati:

decimale	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

C.23 Convertite il valore binario `100101111010` in ottale e in esadecimale.

C.24 Convertite il valore esadecimale `3A7D` in binario.

C.25 Convertite il valore esadecimale `765F` in ottale. (Suggerimento: convertite prima `765F` in binario e poi convertite il valore binario in ottale).

C.26 Convertite il valore binario `1011110` in decimale.

C.27 *Convertite il* valore ottale `426` in decimale.

C.28 Convertite il valore esadecimale `FFFF` in decimale.

C.29 Convertite il valore decimale `299` in binario, ottale ed esadecimale.

C.30 Mostrate la rappresentazione binaria del decimale `779`, quindi mostrate il complemento a uno di `779` e il complemento a due di `779`.

C.31 Cosa si otterrà aggiungendo a un numero il suo complemento a due?

C.32 Mostrate il complemento a due del valore intero -1 su una macchina con interi a 32 bit.

APPENDICE D

Il C++ su Internet

Segue una lista di risorse degne di nota sul C++ che sono reperibili su Internet. Abbiamo considerato FAQ (Frequently Asked Questions, cioè le domande più comuni), tutorial, informazioni sullo standard ANSI/ISO, informazioni sui compilatori C++ più utilizzati e informazioni su come reperire gratuitamente alcuni compilatori, demo, libri, tutorial, *software, articoli, interviste, conferenze, giornali e riviste, corsi online e newsgroup.*

Per informazioni sull'ANSI (American National Standards Institute) o per acquistare i documenti degli standards, visitate il sito dell'ANSI all'URL http://www.ansi.org.

D.1 Risorse

`http://www.progsource.com/index.html`

Ricca collezione di risorse per reperire informazioni su diversi linguaggi di programmazione, tra cui il C++. Include un elenco di strumenti, compilatori, software, testi e altre risorse sul C++.

`http://sunir.org/booklist/#C++`

"Programmer's Book List", *elenco di testi per i programmatori: include una sezione sul* C++ con più di 30 titoli.

`http://www.vcdj.com`

"Visual C++ Developer's Journal", Il giornale del programmatore Visual C++, include articoli recenti, informazioni sulle conferenze, una banca lavoro, demo e altre risorse.

`http://www.genitor.com/resources.htm`

"Developer Resources", Risorse per i programmatori: include link a compilatori C++, utili strumenti per lavorare in C++ e codici sorgenti pubblicati su "C/C++ Users Journal".

`http://www.possibility.com/Cpp/CppCodingStandard.html`

"C++ Coding Standard", Standard di programmazione in C++: include una notevole quantità di informazioni sul C++ e un ricco elenco di altre risorse in rete.

D.2 Tutorial

`http://www.icce.rug.nl/docs/cplusplus/cplusplus.html`

Questo tutorial, scritto da un professore universitario, si rivolge ai programmatori C che vogliono passare al C++.

`http://www.rdw.tec.mn.us/`

Il "Red Wing/Winona Technical College" offre corsi online sul C++.

`http://www.zdu.com/zdu/catalog/programming.htm`

La "ZD Net University" offre diversi corsi online che sono correlati al C++.

`http://library.advanced.org/3074/`

Questo tutorial si rivolge ai programmatori Pascal che vogliono imparare il C++.

`http://www.cpp-programming.com/`

Questo tutorial si rivolge a programmatori di qualsiasi livello, novizi o esperti, che utilizzano le piattaforme DOS o Windows. Purtroppo non è disponibile un indice dei contenuti che permetta di valutare le informazioni che contiene.

`ftp://rtfm.mit.edu/pub/usenet/news.answers/C-faq/learn-c-cpp-today`

Questo sito include un elenco di tutorial sul C++, con la descrizione di ognuno di essi. Vi troverete anche informazioni sulle origini del C e del C++, insieme con informazioni sui diversi compilatori C++ esistenti per le varie piattaforme.

D.3 FAQ (risposte alle domande più comuni)

`http://reality.sgi.com/austern/std-c++/faq.html`

Questo sito di FAQ affronta i vari interrogativi sullo standard ANSI/ISO, sulla progettazione del linguaggio C++ e sulle modifiche più recenti alle caratteristiche del linguaggio.

`http://www.cerfnet.com/~mpcline/C++-FAQs-Lite/`

Questo sito abbonda di FAQ suddivise in 35 categorie. Vi troverete numerose informazioni utili sul C++. Alcune risposte includono sorgenti di esempio.

D.4 comp.lang.c++

`http://weblab.research.att.com/phoaks/comp/lang/c++/resources0.html`

Uno dei siti più ricchi in assoluto di risorse e informazioni di news:comp.lang.c++. Il titolo della pagina, "People Helping One Another Know Stuff" (Diamoci una mano a diventare esperti!), è un indizio di quello che è possibile trovarci. Vi troverete link a oltre 40 risorse per reperire informazioni sul C++.

`http://kom.net/~dbrick/newspage/comp.lang.c++.html`

Visitate questo sito per leggere i newsgroup della gerarchia news:comp.lang.c++.

`http://www.austinlinks.com/CPlusPlus/`

Sito web di "Quadralay Corporation": include link a risorse sul C++ tra cui le librerie Visual C++/MICROSOFT FOUNDATION CLASSES, informazioni sulla programmazione in C++, informazioni lavorative sul C++ e un elenco di tutorial e altri strumenti online per imparare il C++.

`http://db.csie.ncu.edu.tw/~kant_c/C/chapter2_21.html`

Questo sito web contiene un elenco di funzioni della libreria standard dell'ANSI C.

`http://www.csci.csusb.edu/dick/c++std/`

Questo sito include link allo standard ANSI/ISO e al gruppo di discussione di Usenet news:comp.std.c++, che contiene informazioni recenti sullo standard.

`http://ibd.ar.com/ger/comp.lang.c++.html`

"Green Eggs Report": elenca oltre 100 URL relativi a news:comp.lang.C++.

`http://www.research.att.com/~bs/homepage.html`

Homepage di Bjarne Stroustrup, progettista del C++. Fornisce un elenco di risorse sul C++, di FAQ e di altre utili informazioni.

`news:comp.lang.c++`

Gruppo di discussione sul C++.

`news:comp.lang.c++.leda`

Gruppo di discussione sulla libreria di strutture dati e algoritmi LEDA.

`news:comp.lang.c++.moderated`

Discussioni tecniche sul linguaggio C++.

D.5 Compilatori

`http://www.progsource.com/index.html`

"Programmer's Source" (Fonte del programmatore) è una notevole fonte di informazioni su diversi linguaggi di programmazione, tra cui il C++. Vi troverete elenchi di strumenti, compilatori, software, testi e altre risorse. L'elenco di compilatori è organizzato per piattaforma.

`http://www.cygnus.com/misc/gnu-win32/`

L'ambiente di sviluppo GNU è disponibile gratuitamente sul sito di Cygnus.

`http://msdn.microsoft.com/visualc/`

Homepage di Microsoft Visual C++: include informazioni sul prodotto, commenti, materiale supplementare e informazioni su come ordinare il compilatore Visual C++.

`http://www.borland.com/bcppbuilder/freecompiler/`

La demo di "Borland C++ 5.5 " si può scaricare gratuitamente. Per ordinare il prodotto bisogna seguire le indicazioni fornite.

`http://www.borland.com/borlandcpp/turbosuite/`

Sito web del compilatore "Borland Turbo C++ Visual Edition" per Windows.

`http://www-4.ibm.com/software/ad/vacpp/`

Homepage di "IBM VisualAge for C++"

`http://www.metrowerks.com/products/`

Metrowerks CodeWarrior per Macintosh e altri sistemi operativi.

D.6 Standard Template Library

Tutorial

`http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html`

Questo tutorial è organizzato per esempi, filosofia, componenti ed estensioni alla STL. Troverete esempi di codice che utilizzano i componenti della STL, utili spiegazioni e diagrammi.

`http://web.ftech.net/~honeyg/articles/eff_stl.htm`

Questo tutorial offre delle informazioni sui componenti della STL, i container, gli adattatori di stream e gli iteratori, la trasformazione e la selezione di valori, il filtraggio e la trasformazione dei valori e gli oggetti.

`http://www.stanford.edu/cgi-bin/leland/~iburrell/cpp/stl.html`

Questo sito contiene dei link alle risorse sulla STL tra cui FAQ, siti FTP e tutorial.

`http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html`

Questo sito è utile per chi sta comincia a studiare la STL. Si trova un'introduzione alla STL e gli esempi dell'ObjectSpace STL Toolkit.

Riferimenti

`http://www.sgi.com/Technology/STL/other_resources.html`

Questo sito contiene una lista di oltre 15 siti web sulla STL e una lista di letture consigliate sulla STL.

`ftp://ftp.blueneptune.com/pub/users/yotam/stlqr-1.01.tar.gz`

`ftp://ftp.blueneptune.com/pub/users/yotam/stlqr101.zip`

"STL-Quick-Reference version 1.01"

`http://www.cs.rpi.edu/projects/STL/stl/stl.html`

Questa è la homepage della STL. Si trovano spiegazioni dettagliate sulla STL e link ad altre utili risorse.

`http://www.sgi.com/Technology/STL/`

"Silicon Graphics Standard Template Library Programmer's Guide": utile risorsa per la STL. Si può scaricare la STL da questo sito, trovare le informazioni più recenti, la documentazione di progetto e i link ad altre risorse.

FAQ

`http://www.stanford.edu/~iburrell/cpp/stl.html`

Questo sito contiene link a risorse sulla STL tra cui FAQ, siti FTP e tutorial.

`ftp://butler.hpl.hp.com/stl/stl.faq`

Questo sito FTP è un FAQ per la STL mantenuto da Marian Corcoran, membro dell'ANSI Committee ed esperta di C++.

Articoli, libri e interviste

`http://www.sgi.com/Technology/STL/other_resources.html`

Questo sito contiene una lista di oltre 15 siti web sulla STL e una breve lista di libri consigliati sulla STL.

`http://www.byte.com/art/9510/sec12/art3.htm`

Il sito di "Byte Magazine" contiene una copia di un articolo sulla STL scritto da Alexander Stepanov. Stepanov, uno dei creatori della Standard Template Library, fornisce informazioni sull'uso della STL nella programmazione generica.

`ftp://ftp.cs.rpi.edu/pub/stl/`

Documentazione e ricerche sull'implementazione di Hewlett-Packard della STL (Standard Template Library). Questo sito include anche il codice sorgente per STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library di D.R. Musser e Atul Saini, Addison-Wesley, Reading, MA, 1996.

`http://www.sgi.com/Technology/STL/drdobbs-interview.html`

Questa intervista con Stepanov contiene informazioni interessanti sulla creazione della Standard Template Library. Stepanov parla di come sono stati concepiti la STL, la programmazione generica, l'acronimo "STL" e altro ancora.

Software

`http://www.cs.rpi.edu/~musser/stl.html`

Il sito RPI STL include informazioni sulle differenze tra STL e altre librerie C++, su come compilare i programmi che utilizzano la STL, contiene una lista dei principali file di intestazione della STL, programmi di esempio che utilizzano la STL, le classi container della STL e le categorie di iteratori della STL. Contiene anche una lista di compilatori compatibili con la STL, di siti FTP di codice sorgente della STL e di materiale correlato.

`ftp://ftp.cs.rpi.edu/pub/stl/`

Documentazione e ricerche sull'implementazione di Hewlett-Packard della STL (Standard Template Library). Questo sito include anche il codice sorgente per "STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library" di D.R. Musser e Atul Saini, Addison-Wesley, Reading, MA, 1996.

`http://www.mathcs.sjsu.edu/faculty/horstman/safestl.html`

Scaricate SAFESTL.ZIP, uno strumento progettato per trovare gli errori nei programmi che utilizzano la STL.

`http://www.sirius.com/~ouchida/`

Silicon Graphics, Inc. STL per Microsoft Visual C++ 5.0.

`http://www.cs.rpi.edu/~wiseb/stl-borland.html`

"Using the Standard Template Library with Borland C++." Questo sito è un riferimento utile per chi usa il compilatore Borland C++. L'autore ha sezioni sugli avvertimenti e sulle incompatibilità.

`http://www.geocities.com/SiliconValley/Pines/2010/note.txt`

Contiene le incompatibilità tra STL e Microsoft Developer's C++ 4.2.

Indice analitico

Simboli

--, 72
- meno (unario), 633
-- postdecremento (unario), 633
- sottrazione, 634
!, 97
! (NOT logico), 95
! negazione logica (unario), 633
!=, 29
!= relazionale, diverso da, 634
#define NDEBUG, 428
#endif, 361
#ifndef, 361
#include, 146, 358
%, 66
% modulo, 634
%=, 72
%= assegnamento con modulo, 634
& AND su bit, 634
& *indirizzo*, 633
&&, 97
&& (AND logico), 95
&& AND logico, 634
&= assegnamento con AND su bit, 634
(type) cast unario in stile C, 633
() parentesi, 633
punto e virgola (, 17
*, 66
* indirezione, 633
* moltiplicazione, 634
*, operatore di risoluzione del riferimento, 266
*/, 16
*=, 72
*= assegnamento con moltiplicazione, 634
, virgola, 634
. selezione del membro tramite un oggetto, 633
.* puntatore al membro tramite un oggetto, 633
.C, 12
.cpp, 12
.cxx, 12
.h, 17, 144
/, 66
/ divisione, 634
/*, 16
/=, 72
/= assegnamento con divisione, 634
:: *risoluzione dello scope (binario)*, 633
:: risoluzione dello scope (unario), 633
?: condizionale (ternario), 634
[] indicizzazione di array, 633
^ OR esclusivo su bit, 634
^= assegnamento con OR esclusivo su bit, 634
| OR inclusivo su bit, 634
|| (OR logico), 95
|| OR logico, 634
|= assegnamento con OR inclusivo su bit, 634
~ complemento a bit (unario), 633
'\n', 303
+ addizione, 634
+ più (unario), 633
++, 72
++ postincremento (unario), 633
+=, 72
+= assegnamento con addizione, 634
< relazionale, minore di, 634
<< shifting a sinistra dei bit, 634
<<= assegnamento con shifting a sinistra su bit, 634
<=, 29
<= relazionale, minore di, 634
<*algorithm*>, 145
<assert.h>, 144
<bitset>, 145
<cctype>, 144
<cfloat>, 144
<climits>, 144
<cmath>, 144
<cstdio>, 145
<cstdlib>, 145
<cstring>, 145

<ctime>, 145
<ctrl>-d, 605
<ctrl>-z, 605
<ctrl-d>, 88
<ctrl-z>, 88
<ctype.h>, 144
<deque>, 145
<exception>, 145
<float.h>, 144
<fstream.h>, 145, 597
<fstream>, 145
<functional>, 145
<iomanip.h>, 145, 597, 610
<iomanip>, 145
<iostream.h>, 145, 597
<iostream>, 145
<iterator>, 145
<limits.h>, 144
<limits>, 146
<list>, 145
<locale>, 146
<map>, 145
<math.h>, 144
<memory>, 145
<queue>, 145
<set>, 145
<sstream>, 146
<stack>, 145
<stddef.h>, 305
<stdexcept>, 145
<stdio.h>, 145
<stdlib.h>, 145
<string.h>, 145, 305
<string>, 146
<time.h>, 145
<typeinfo>, 146
<utility>, 145
<vector>, 145
-=, 72
= assegnamento, 634
-= assegnamento con sottrazione, 634
==, 29
== relazionale, uguale a, 634
> relazionale, maggiore di, 634
-> selezione del membro tramite un puntatore, 633
->* puntatore al membro tramite un puntatore, 634
>=, 29
>= relazionale, maggiore di o uguale a, 634
>> shifting a destra dei bit, 634
>>= assegnamento con shifting a destra su bit, 634
0 e 1, sequenze di, 6
0, puntatore, 266
0x, 620
0X, 620

A

a.out, 12
abort come funzione di default chiamata da assert, 428
accesso ai membri di una classe, 356
accodamento, 431
Ada, 11
addizione di due numeri interi, 20
ADT, 429
aggiunta di nuove classi a una gerarchia, 579
albero, 265
algoritmi di ordinamento, 227
algoritmo, 47
algoritmo di mescolamento, 294
algoritmo di ordinamento a bolle con i puntatori, 279
ambiente C++, elementi fondamentali, 11
ambienti C/C++ integrati, 3
ambiguità, 552
ampiezza dei campi, 612
An90, 14
analisi orientata agli oggetti (OOA), 111
analytical engine, 11
ANSI (*American National Standards Institute*), 2
Apice doppio, 18
Apple Computer, 6
approccio a blocchi, 9
approccio *live-code*, 1
argomento di default, 177
aritmetica dei puntatori, 285
array, 205
array bidimensionale, 238
array che, 431
array che memorizza una stringa di caratteri, 218
array di puntatori, 292
array di stringhe suit, 292
array multidimensionale, 238
array per riepilogare dati di un sondaggio, 213
array static, 220
array, dichiarazione di un, 208
array, elemento di un, 205
array, indice, 206
array, inizializzatore, 208
array, numero posizionale di un elemento, 205
array, oltrepassare i limiti di un, 215
array, passaggio di un - a una funzione, 222
array, primo elemento numero zero, 206
array, valori di inizializzazione, 208
arrotondamento, 25
ASCII, 310
assegnamento di due array, 430
assegnamento di puntatori di classe base e derivata, 527
assegnamento tra oggetti, 382
assembler, 7
assembly, 6
assert, 428
assert.h, 428
asserzione, 428
associatività degli operatori, 26
associazione, 114, 541
astrazione, 518
astrazione dei dati, 35, 343, 399, 429
attivazione e disattivazione dei flag di formattazione, 622
attributi, 35
attributo, 184
attributo di stato, 559
attivazione e disattivazione, 559
attributo di una classe, 113
auto, 155
autoassegnamento, 418, 488
autodocumentante, 21
azione, 187
azioni, 2

B

B, 8
Babbage, Charles, 11
badbit, 603, 624
Barra inversa, 18
barra rovesciata, 17
base dei numeri interi, 620
base dei numeri interi su uno stream, 610
batch, 5
BCPL, 8
Bell Laboratories, 8
binding differito, 579
binding dinamico, 565, 578, 579
binding statico, 565
bit di stato, 603
blocco, 57
blocco esterno, 179
blocco interno *vede* solo identificatore locale, 158
bool, 52, 96
Borland C++, 12
break, 88, 93
bubble sort, 227
buffer, 626
buffer di output, 23
buffering, 626

C

C, 2
C classico, 8
C di Kernighan e Ritchie, 8
C tradizionale, 8
C++ è linguaggio essenziale, 431
C++ fa differenza tra lettere maiuscole e minuscole, 21
C++, evoluzione naturale del C, 8
calcoli aritmetici, 25
campanello, 107
Campanello, 18
campo di tipo, 563
campo giustificato, 619
Carattere, 18
carattere alfanumerico, 21
carattere di escape, 17
carattere di nuova linea, 600
carattere di riempimento, 612, 619

carattere di sottolineatura (_), 21
carattere due punti (:) segnala ereditarietà, 526
carattere EOF, 606
carattere nullo che segnala la fine di una stringa, 218
caratteri e stringhe, 303
caricamento, 11
case, 85
caso base, 161
caso d'uso di un sistema, 111
cast di un puntatore a una classe base in puntatore a una classe derivata, 521
CD-ROM, 4
cerchietti, 49
cerr, 14, 597
char, 21
chiamata di funzione ha un costo in termini di tempo di elaborazione e di memoria, 170
chiamata implicita a un costruttore di default, 552
chiamata per indirizzo, 173
chiamata per riferimento, 173
chiamata per riferimento con argomenti di tipo puntatore, 269
chiamata per riferimento migliora le prestazioni di un programma, 173
chiamata per valore, 173
chiamata ricorsiva, 161
chiarezza, 14
cicli controllati da variabili contatore, 75
ciclo di simulazione, 249
cifre, 637
cin, 13, 597
cin.eof(), 606
cin.get, 607
cin.getline, 304
cin.ignore, 436
cin.tie, 626
class, 182
classe, 9, 35, 133, 343, 399
classe Array, 477
classe astratta definisce interfaccia per i membri di una gerarchia di classi, 567
classe base, 517, 519
classe base astratta, 565
classe base diretta, 533
classe base indiretta, 533
classe base ios, 597
classe concreta, 565
classe container, 432
classe Date, 503
classe derivata, 517, 519
classe Employee, 413
classe istream, 475, 597
classe ostream, 475, 597
classe proxy, 432
classe String, 490
classe, file di intestazione di una, 358
classe, implementazione di una - nascosta ai client, 352
classe, inizializzazione dei dati-membro di una, 353
classe, modalità di default relativa all'accesso ai membri è private, 363
classe, nome di una, 350
classi astratte nei primi livelli della gerarchia, 566
classi rendono C++ estensibile, 350
classi semplificano le strutture di controllo di un programma, 367
client, 349
client/server, 6
clog, 597
cmath, 135
COBOL, 10
coda, 265
codice oggetto, 12
codici di carattere, 310
codici numerici dei caratteri, 309
codifica dei nomi, 181
collaborazione, 185, 312
collegamento, 116
collegamento di uno stream di output a uno stream di input, 626
collezione, 432
commenti, 16
commento su linea singola, 16
comp.lang.c++, risorse del newsgroup in rete, 650
compilatore, 7
compilatori C++ in rete, 651

compilazione, 11
complemento a uno, 645
complessità esponenziale, 167
componente software, 9
comportamenti, 35
comportamento, 244
composizione, 355, 408
composizione e software riutilizzabile, 408
composizione ed ereditarietà, 540
computer, 1, 3
concetti fondamentali dei computer, 1
condivisione delle risorse, 5
condizione, 28
condizione di guardia, 187
confondere l'operatore di uguaglianza == con l'operatore di assegnamento =, 98
confronto tra due array, 431
const, 172, 211, 399
const_cast< type > eliminazione del carattere const, 633
contatore, 58
continue, 93
controllo del programma, 48
controllo dell'accesso ai membri di una classe, 361
controllo della validità degli indici, 430
controllo delle lettere maiuscole/minuscole, 622
conversione da binario a decimale, 642
conversione da decimale a binario, 643
conversione da decimale a esadecimale, 643
conversione da decimale a ottale, 643
conversione da esadecimale a binario, 642
conversione da esadecimale a decimale, 642, 643
conversione da ottale a decimale, 642, 643
conversione da ottale in binario, 642
conversione di puntatori, 527
conversione esplicita, 65
conversione forzata degli argomenti, 142
conversione implicita, 66
conversioni implicite di oggetti, 537
conversioni tra tipi diversi, 489
copia dei valori in una chiamata per valore, 270
copia di default membro a membro, 382, 486
costante di enumerazione, 154
costante nominale, 211
costante stringa, 303
costruttore, 349, 367
costruttore di conversione, 489
costruttore di copia, 484, 486
costruttore di default, 368
costruttore non restituisce alcun tipo di dato, 350
costruttori e argomenti di default, 368
costruttori e distruttori nelle classi derivate, 534
cout, 13, 597
cout.rdstate, 624

D

dati, 4
dato membro, 344, 348
DEC PDP-7, 8
decisioni, 2
default, 89
definizione, 75
delete deallocazione dinamica della memoria, 633
delete[] deallocazione dinamica di array, 633
delimitatore, 304, 310, 607
derivare una classe da più classi base, 548
dettagli di implementazione, 343
diagramma degli oggetti, 116
diagramma dei casi d'uso, 111
diagramma delle attività, 188
diagramma delle classi, 113, 184
diagramma delle collaborazioni, 314
diagramma di flusso, 49
diagramma di sequenza, 249
diagramma di stato, 186
dichiarazione, 21
dichiarazione anticipata, 391, 434, 446
dichiarazione di un costruttore di conversione, 498
dichiarazione di una funzione friend, 413
differimento indefinito, 294
Digital Equipment Corporation, 88

dimensione, 24
Dipartimento di Difesa degli Stati Uniti (DoD), 11
direttiva al preprocessore, 12
direttive condizionali, 361
disco, 5
dispositivo di input standard, 597
dispositivo di output standard, 598
distruttore, 353, 371
distruttore di default, 353
distruttore non prende argomenti, 353
distruttore virtuale, 579
distruttori chiamati in ordine inverso rispetto ai costruttori, 372
divide et impera, 133
divisione intera, 25
divisione per zero, 14
do/while, 50
documentare i programmi, 16
double, 83
dynamic_cast< type > cast con controllo di tipo a run-time, 633

E

EBCDIC, 310
effetti collaterali, 166
elaborazione batch, 5
elaborazione interattiva, 23
eliminazione dei goto, 49
emacs, 12
endl, 23, 600
entità statica, 205
enum, 151
EOF, 304, 606, 608
eofbit, 624
ereditarietà, 35, 355, 517, 553
ereditarietà di implementazione, 580
ereditarietà di interfaccia, 580
ereditarietà e ingegneria del software, 539
ereditarietà multipla, 517, 548
ereditarietà multipla e software riutilizzabile, 548
ereditarietà private, 518, 532
ereditarietà protected, 518, 532
ereditarietà public, 518, 532
ereditarietà singola, 517
errore di sintassi, 18
errore logico, 56
errore logico fatale, 56
errore logico non fatale, 56
errore logico off-by-one, 77
esecuzione condizionale delle attività, 250
esecuzione sequenziale, 48
esecuzione, fase di, 11
esercizi avanzati sulla manipolazione di stringhe, 338
esercizi sulla manipolazione di stringhe, 336
espressione condizionale, 54
espressione mista, 143
etichetta, 158
etichetta di azione, 188
etichetta di azione exit, 188
etichetta di una struttura, 344
evento, 186
evitare classi inutili, 539
extern, 155

F

failbit, 603, 609, 624
fallimento di new, 422
false, 28, 52
FAQ sul C++ in rete, 650
fattore di scala, 147
fattoriale, 162
fattoriale, definizione ricorsiva del, 162
Fibonacci, 164
file di intestazione, 16, 579
file oggetto, 579
file server, 6
fine del file (EOF), 304
flag, 603
flag di formattazione, 615
flag internal, 618
flag ios::dec, 620
flag ios::fixed, 621
flag ios::hex, 620
flag ios::oct, 620
flag ios::scientific, 621
flag ios::uppercase, 622
flag left, 617
flag right, 617
flag showpoint, 616
float, 21

flush, 609
flusso. *Vedi* stream
flusso di controllo, 32
flusso di controllo del programma, 57
flusso di controllo di una chiamata a una funzione virtuale, 590
for, 50
formato fixed, 621
formato scientific, 621
formattazione a virgola fissa, 66
FORTRAN, 10
free, 421
friend, 350, 413
funzione, 9, 17, 133
funzione ascending, 298
funzione chiamante, 134
funzione chiamata, 134
funzione della libreria standard, 133
funzione descending, 298
funzione di accesso, 363, 364
funzione di utilità, 364
funzione fibonacci, 165
funzione friend, 350, 413
funzione generica, 182
funzione get, 363, 375, 582
funzione in linea, 171
funzione in linea cube, 172
funzione membro, 36, 344, 348
funzione membro all'interno della definizione di una classe è inline, 354
funzione membro bad, 624
funzione membro clear, 624
funzione membro const, 399
funzione membro fail, 624
funzione membro fill, 617, 619
funzione membro flags, 615, 622
funzione membro gcount, 609
funzione membro get, 606
funzione membro getline, 607
funzione membro good, 624
funzione membro ignore, 608
funzione membro inline, 358
funzione membro operator!, 624
funzione membro peek, 608
funzione membro precision, 610, 621
funzione membro put, 602
funzione membro putback, 608
funzione membro rdstate, 624
funzione membro read, 609
funzione membro setf, 615
funzione membro unsetf, 615, 623
funzione membro write, 609
funzione operatore commutativa, 472
funzione predicativa, 364
funzione ricorsiva, 161
funzione ricorsiva factorial, 163
funzione rollDice, 153
funzione set, 363, 375, 576, 582
funzione virtuale, 563, 564
funzione virtuale pura, 565
funzione width, 612
funzione, argomento di una, 134
funzione, chiamata di, 134
funzione, corpo di una, 139
funzione, definizione, 137
funzione, definizione che funge da prototipo, 138
funzione, invocare una, 134
funzione, lista di parametri di una, 139
funzione, tipo di dato restituito da una, 138
funzioni definite dal programmatore, 134
funzioni della libreria standard per la manipolazione delle stringhe, 302
funzioni di interrogazione, 375
funzioni di query, 375
funzioni friend non sono ereditate, 521
funzioni matematiche, 135
funzioni matematiche trattano argomenti double, 135
funzioni membro di classe base, 527
funzioni set/get isolano implementazione di una classe, 380
funzioni, overloading di, 180
funzioni, relazioni gerarchiche tra, 134

G

generazione di numeri casuali, 146
gerarchia Point, Circle e Cylinder, 580
giochi, 146
giochi d'azzardo, 151
giochi d'azzardo, 146
gioco crap, 151
giustificazione, 617

goodbit, 624
goto, 49

H

handle, 356, 386
handle di un oggetto, 356
handle implicito di una funzione membro, 413
hardware, 2, 4

I

I/O *type-safe*, 608
I/O del C++ è *type safe*, 595
I/O formattato, 597
I/O in stile C, 602
I/O non formattato, 597, 609
IBM, 6
, 6
identificatore, 21
identificatore esterno, 157
if, 29, 50
if/else nidificati, 54
immagine eseguibile, 12
implementazione del tipo di dato Time come classe, 348
implementazione del tipo di dato Time come struttura, 346
implementazione di polimorfismo, funzioni virtuali e binding dinamico, 588
implementazione di una classe proxy, 433
incapsulamento, 343
indici a scelta, 430
informazioni di collegamento, 155
informazioni di memorizzazione, 155
ingegneria del software, 9
inizializzatore = 0, 565
inizializzatore di membro, 408
inizializzatore di un oggetto const, 404
inizializzazione degli oggetti di una classe, 367
inline, 171, 354
input, 4
input da stream, 603
input e output di array, 431
input/output su stream, 597
insieme dei caratteri ASCII, 635
insieme di caratteri ASCII, 635
int, 21
int possono avere dimensione diversa su sistemi diversi, 91
interfaccia, 35, 343
interfaccia della classe, 349
interfaccia pubblica, 568
interfaccia pubblica della classe, 353
Internet, 3
intero, 21
interprete, 7
intestazione di for, 82
ios::dec, 615
ios::fixed, 66, 615
ios::hex, 615
ios::internal, 615
ios::left, 615
ios::oct, 615
ios::right, 615
ios::scientific, 615
ios::showbase, 615
ios::showpoint, 66, 615
ios::showpos, 615
ios::skipws, 615
ios::uppercase, 615
iostream.h, 16
ISO (*International Standards Organization*), 2
istanziare, 344
istogramma, 216
istruzione, 17
istruzione composta, 55
istruzione return, 138
istruzione switch, 563
iteratore, 432
iteratore normalmente friend, 432
iterazione / ricorsione, 168
iterazione controllata da un contatore, 58
iterazione controllata da un valore sentinella, 60

J

Java, 3

L

label, 158
LAN, 6

late binding, 579
leggibilità, 21
libreria iostream, 597
libreria standard, 9, 133
linea di evoluzione di un oggetto, 249
linea di flusso, 49
linea di testo, programma per visualizzare, 16
linguaggi procedurali, 35
linguaggio, 1
linguaggio ad alto livello, 6, 7
linguaggio assembly, 6
linguaggio di programmazione procedurale, 344
linguaggio macchina, 6
linker, 12
linking, 11
lista concatenata, 265
lista di inizializzatori di membro, 498
lista di parametri vuota, 170
literal, 22
locazione, 24
long, 91
long int, 91
lunghezza della parola, 430
lvalue, 99, 175, 206

M

macro, 144
macro assert, 428, 531
main, 17, 134
main implementata come elenco di chiamate a funzioni, 136
malloc, 421
manipolare strutture di dati dinamiche, 265
manipolatore bell (beep), 614
manipolatore dec, 610
manipolatore dello stream setw, 304
manipolatore di stream, 600, 609
manipolatore di stream endl, 23
manipolatore di stream non parametrizzato, 66
manipolatore di stream parametrizzato, 66, 597
manipolatore di stream parametrizzato setfill, 617
manipolatore di stream parametrizzato setiosflags, 616
manipolatore endLine (fine linea), 614
manipolatore hex, 610
manipolatore oct, 610
manipolatore ret (a capo), 614
manipolatore setbase, 610
manipolatore setprecision, 610
manipolatore tab (tabulazione), 614
manipolatori definiti dall'utente, 614
manipolazione di caratteri e stringhe, 302
manipolazione di oggetti, 2
manipolazione di stringhe, 265
math.h, 83, 135
matrice, 238
maximum, 140
meccanismi decisionali, 4
meccanismo di gestione degli arrivi, 106
media, 229
media aritmetica, 26
media dei voti ricevuti dagli studenti di una classe, 58
mediana, 229
membro, 344
membro protected, 521
membro static di una classe, 423
membro statico ios::basefield, 620
memoria, 4
memoria primaria, capacità bassa, 4
memorizzazione automatica, 155
memorizzazione automatica risparmia l'utilizzo della memoria, 156
memorizzazione statica, 155
messaggio, 245
messaggio *write*, 568
metodo, 35, 344
metodo di accesso, 363
Microsoft Visual C++, 12
Microsoft Windows, 6
miglioramenti del C++ al C, 3
moda, 229
modellare gli oggetti del mondo reale, 35
molteplicità, 114
moltiplicazione (*), 25
mouse, 4
multiprogrammazione, 5
multitasking, 11

N

new, 531
new allocazione dinamica della memoria, 633
new[] allocazione dinamica di array, 633
nidificazione, 51
nome di funzione, 157
nome di un array, 278
nome di un array è puntatore, 288
nome di un tipo di dato, 154
nome di variabile, 21, 266
nomi presenti nella definizione di un problema, 112
notazione con complemento a due, 644
notazione posizionale, 638
notazione puntatore/offset, 288
notazione scientifica, 66
NULL, 266
numeri a virgola mobile e notazione scientifica, 621
numeri a virgola mobile, imprecisi, 67
numeri binari negativi, 644
numeri casuali, 146
numeri magici, 213
numeri pseudocasuali, 149
numero aureo, 164
numero reale, 21
nuova linea (\n), 17
nuove convenzioni, 32

O

occultamento delle informazioni, 158, 343, 429
offset, 288, 590
oggetti dovrebbero essere sempre in uno stato coerente, 486
oggetto, 9
oggetto client, 245
oggetto const, 399
oggetto contiene soltanto dati, 355
oggetto della classe derivata è oggetto della classe base, 518
oggetto *host*, 408
oggetto iteratore, 432
oggetto membro, 408
oggetto server, 245
OOP, 343
operando destro, 17
operator!=, 489
operator<<, 475
operator=, 488
operator>>, 475
operatore &, 267
operatore << con stringhe di caratteri, 218
operatore >> con stringhe di caratteri, 219
operatore binario, 23
operatore binario di risoluzione dello scope (
), 423
operatore binario di risoluzione dello scope (::), 354
operatore cast su puntatori, 287
operatore condizionale (?:), 54
operatore delete, 421, 579
operatore di accesso al membro, 345
operatore di assegnamento (=) non ha bisogno di overloading, 469
operatore di assegnamento =, 23
operatore di assegnamento con addizione +=, 71
operatore di cast, 63, 143, 490
operatore di cast unario, 65
operatore di conversione, 490
operatore di dereferenza, 267
operatore di estrazione dallo stream, 603
operatore di estrazione dallo stream (>>), 597
operatore di estrazione dallo stream, >>, 20
operatore di inserimento nello stream (<<), 597, 599
operatore di inserimento nello stream <<, 17
operatore di risoluzione dello scope, 179
operatore esponente, 27
operatore freccia (->), 345, 418
operatore indirizzo (&), 267
operatore indirizzo (&) non ha bisogno di overloading, 469
operatore modulo (%), 25
operatore new, 421, 531

operatore punto (.), 345, 418
operatore sizeof, 284
operatore ternario (?
), 54
operatore unario di risoluzione dello scope (::), 179
operatori aritmetici, 25
operatori di assegnamento, 71
operatori di assegnamento aritmetici, 72
operatori di incremento e decremento, 72
operatori di inserimento/estrazione in cascata, 601
operatori di uguaglianza, 28
operatori logici, 95
operatori relazionali, 28
operatori, ordine di valutazione degli - non specificato, 166
operazione, 184, 244
operazione di una classe, 113
operazioni consentite su un dato, 430
operazioni di input/output, 595
operazioni matematiche, 3
orario, formato militare, 346
orario, formato standard, 346
orario, formato universale, 346
ordinamento, 227
ordinamento a bolle, 227
ordinamento lessicografico, 309
ordine delle azioni è importante, 47
ordine di chiamata dei costruttori e dei distruttori, 372
ordine di chiamata di costruttori e distruttori, 534
output delle variabili di tipo char *, 602
output di caratteri tramite put, 602
output di una stringa tramite un'istruzione di inserimento nello stream, 599
output su stream, 599
ovali, 49
overflow, 430
overloading degli operatori, 468
overloading degli operatori consente notazione concisa, 468
overloading degli operatori di inserimento/estrazione per l'I/O su stream, 473
overloading degli operatori rende C++ estensibile, 468
overloading degli operatori, restrizioni, 470
overloading dell'operatore <<, 472
overloading dell'operatore di complementazione, 500
overloading dell'operatore di disuguaglianza (!=), 489
overloading dell'operatore unario !, 476
overloading di —, 502
overloading di ++, 502
overloading di un operatore binario, 476
overloading di un operatore unario, 476
overloading di una funzione, 180
overloading inappropriato, 469
overloading non influisce sul numero di operandi di un operatore, 471
overloading non influisce sulla precedenza di un operatore, 470
overloading non influisce sull'associatività di un operatore, 470
overriding, 528, 564
overriding delle funzioni di classe base, 532

P

parametro, 136
parametro di riferimento, 173
PARC (Palo Alto Research Center), 9
parentesi angolari, 66
parentesi costituiscono l'ordine massimo di precedenza, 26
parentesi dello stesso livello, 27
parentesi graffa aperta, 17
parentesi graffa chiusa, 17
parentesi per *forzare* l'ordine di calcolo, 26
parola, 430
parola chiave operator, 468
parola riservata, 50
parola riservata class, 182, 348
parola riservata struct, 344
parola riservata template, 182
parte del mondo reale di una simulazione, 110
parte di controllo di una simulazione, 110
Pascal, 2

Pascal, Blaise, 11
passaggio dei parametri, 274
passeggero, 107
passo ricorsivo, 161
pensare in termini di oggetti, 3
Pensare in termini di oggetti, 106
periferica, 4
personal computer, 6
Personal Computer IBM, 6
piattaforma hardware, 8
pila, 265, 429
polimorfismo, 517, 563, 566
polimorfismo promuove l'estensibilità del software; 567
polimorfismo si implementa tramite le funzioni virtuali, 566
pop da una pila, 429
portabilità, 8
postdecremento, 72
postincremento, 72
pow, 27
precedenza di un operatore, 26
precisione, 66
precisione dei valori a virgola mobile, 610
predecremento, 72
preincremento, 72
preprocessing, 11
preprocessore, 12
prestazioni, importanza vitale delle, 169
primo raffinamento, 61
principio del controllo invertito, 568
principio del minimo privilegio, 399
principio del minor privilegio, 156
private, 349
privilegio di accesso, 274
problema del costrutto switch, 563
problemi reali, 133
procedura, 47
processo OOAD, 110
progettazione della classe Array, 477
progettazione della classe Circle, 541
progettazione della classe Cylinder, 541
progettazione della classe Date, 503
progettazione della classe Point, 541
progettazione della classe String, 490
progettazione delle funzioni operatore, 471

progettazione di un libro paga elettronico, 568
progettazione orientata agli oggetti, 106
progettazione orientata agli oggetti (OOD), 184, 343
programma che mescola e distribuisce carte da gioco, 293
programma che simula 6000 lanci di un dado, 147
programma per computer, 5
programma testa o croce, 147
programmatore, 8
programmatori, 4
programmazione, 1
programmazione basata sugli oggetti (OBP), 399
programmazione orientata agli oggetti, 106, 343
programmazione orientata agli oggetti (OOP), 1, 399
programmazione orientata all'azione, 344
programmazione strutturata, 1, 343
programmi, 4
programmi di traduzione, 7
programmi scalabili, 212
programmi, suddivisione in più file, 358
promozione, 66
prototipo di funzione, 138, 141
prototipo di funzione con nomi dei parametri, 142
prototipo di funzione, obbligatorietà, 142, 281
proxy, 432
pseudocodice, 48
public, 349
pulsante esterno, 106
pulsante interno, 107
puntatore, 265
puntatore a funzione, 298
puntatore alla classe base "vede" soltanto porzione di classe base dell'oggetto della classe derivata, 526
puntatore costante a dati costanti, 278
puntatore costante a dati non costanti, 278
puntatore dereferenziato è un lvalue, 268
puntatore non costante a dati costanti, 275
puntatore non costante a dati non costanti, 275
puntatore può essere decrementato (—), 285
puntatore può essere incrementato (++), 285
puntatore this, 413, 416, 471
puntatore, dichiarazione di un -, 265
puntatore, formato di visualizzazione di un, 268
puntatore, sottrazione, 285
puntatori e array, correlazione, 287
puntatori, indicizzazione dei, 288
push su una pila, 429

Q

qualificatore const, 225, 275
qualificatore inline, 171

R

raffinamento, 294
rand, 146
rand produce numeri con uguale probabilità, 147
rand, come scalare il risultato di, 147
rand, prevedibilità dei risultati di, 149
RAND_MAX, 146
randomizzazione, 149
rapporto aureo, 164
rappresentazione di un dato, 430
record, 277
register, 155
regole di promozione, 143
regole di visibilità, 157
regole per comporre programmi strutturati, 102
reinterpret_cast< type > cast per conversioni non standard, 633
relazione, 518
relazione "knows a", 540
relazione "uses a", 540
relazione friend è concessione, 414
relazione friend non simmetrica, 414
relazione friend non transitiva, 414
relazione tutto/parte, 114
requisiti del sistema, 111

restituire un riferimento a un dato membro private, 380
restituire un risultato, 134
resto, 25
rete locale, 6
rettangoli, 49
rettangolo, simbolo di azione, 49
return, 18, 138
ricerca, 233
ricerca binaria, 233
ricerca binaria, efficienza della, 235
ricerca lineare, 233
Richards, Martin, 8
ricorsione, 161
ricorsione / iterazione, 168
ricorsione, caso base della, 161
ricorsione, esempi ed esercizi del testo che la utilizzano, 169
ricorsione, ogni livello di - duplica numero di chiamate, 167
ridefinizione attraverso passi di raffinamento top-down, 61
riempimento dei campi, 619
Riepilogo dei concetti fondamentali della programmazione strutturata, 100
riferimento, 173
riferimento nullo, 603
riferimento pendente, 176
ripetizione definita, 58
ripetizioni indefinite, 61
risoluzione dei problemi, 47
risoluzione del riferimento, 266
Risorse sul C++ in rete, 649
Ritchie, Dennis, 8
riutilizzo del software, 136, 355
rombi, 49
rvalue, 99, 175

S

scalare, 223
scanner, acquisire le immagini con, 4
schede perforate, 5
schermo, 4, 598
scope, 78, 155, 157
scope a livello di classe, 354
scope di classe, 423
scostamento, 288, 590

screen manager, 567, 579
segnatura, 181
seme, 149
separazione di interfaccia e implementazione, 357
sequenza dei messaggi, 314
sequenza di strutture di controllo, 51
serie di Fibonacci, 164
servizio, 114, 361
set di caratteri, 310
set di caratteri ASCII, 303
setioflags, 66
setiosflags, 84
setprecision, 84
setw, 304
sezione di inizializzazione di for, 79
sezione speciale
costruite il vostro computer, 327
short, 91
short int, 91
silicio, 2
simbolo di connessione, 50
Simpletron, 327
simulatore software di ascensore, 106
sintassi dell'inizializzatore di membro, 552
sistema client/server, 6
sistema distribuito, 6
sistema estensibile, 563
sistema numerico, 637
sistema numerico binario, 637, 639
sistema numerico decimale, 637, 639
sistema numerico esadecimale, 637, 640
sistema numerico in base 10, 637
sistema numerico in base 2, 637
sistema numerico ottale, 637, 640
sistema operativo, 5, 568
sistemi di rete, 6
sistemi distribuiti, 6
size_t, 305
sizeof, 284
sizeof memoria occupata in byte, 633
sizeof su un oggetto, 416
Smalltalk, 9
SML, 527
software, 4
software riutilizzabile, 517
sommatoria eseguita con for, 82
sottoclasse, 519
sovraccarico di una funzione, 180
spaziatura, 51
spazio dei nomi, 32
specificatore di accesso ai membri, 349
specificatore private, 349
specificatore public, 349
spia dell'ascensore, 107
sqrt, 135
square, 137
srand, 149
stack, 265
Standard Template Library in rete, 652
static, 155
static_cast, 66
static_cast< type > cast con controllo di tipo durante la compilazione, 633
std, 33
stdlib.h, 428
strcat, 305
strcmp, 306
strcpy, 305
stream, 13, 596
stream di dati standard per gli errori, 14
stream di input/output, 16
stringa letterale, 303
stringa, lunghezza di una, 311
stringhe, confronto tra, 307
strncat, 306
strncmp, 306
strncpy, 306
strtok, 306
struct, 344
struttura, 277, 344
struttura dati FIFO, 431
struttura dati LIFO, 429
struttura di controllo, 48
struttura di iterazione, 49
struttura di selezione, 49
struttura di selezione doppia, 50
struttura di selezione if, 50, 51
struttura di selezione if/else, 50, 53
struttura di selezione multipla, 50
struttura di selezione singola, 50
struttura di selezione switch, 50
struttura iterativa, 57
struttura iterativa while, 57

struttura sequenziale, 49
struttura, definizione, 344
struttura, membro di una, 344
struttura, modalità di accesso di default per i membri è public, 363
strutture *a una sola entrata / a una sola uscita*, 51
strutture di controllo nidificate, 67
strutture, problemi dei dati non inizializzati, 348
Sun Microsystem, 10
superclasse, 519
supercomputer, 4
sviluppo rapido di applicazioni (RAD), 384
svuotamento del buffer, 600
svuotare il buffer di output, 23
system box, 111

T

tabella di priorità, 633
tabella di verità dell'operatore && (AND logico), 96
tabelle di verità, 96
Tabulazione orizzontale, 18
tastiera, 4, 597
tasto Invio, 23
template, 182
template di funzione, 182
tenere nascoste le informazioni, 35
terminale, 5
terminatore di istruzione, 17
this, 413
Thompson, Ken, 8
tilde (~), 353, 371
timer, 106
timesharing, 5
tipi di dati astratti (ADT), 429
tipo definito dall'utente, 344
tipo definito dall'utente Time, 346
tipo di dato astratti (ADT), 343
tipo di dato astratto, 430, 431
tipo di dato definito dall'utente, 154
tipo predefinito, 344
tipologie degli algoritmi di iterazione, 58
token, 305
top, 61
totale, 58
transizione tra stati, 186
trasferimento del controllo, 48
trattare un problema in modo generale, 567
troncamento, 25
true, 28, 52
Tutorial sul C++ in rete, 649
type safe, 595
typeid informazioni di tipo a run-time, 633

U

UML, 106
un'espressione condizionale, 52
Unicode, 635
unità a dischi, 4
unità aritmetico-logica (ALU), 4
unità centrale di elaborazione (CPU), 4
unità di input, 4
unità di memorizzazione secondaria, 5
unità di output, 4
UNIX, 6
unsigned, 149
unsigned int, 149
unsigned long, 164
unsigned long int, 164
uscire da una funzione, 18
using namespace std, 32

V

valore, 24
valore chiave, 233
valore di traslazione, 151
valore finale, 75
valore iniziale, 75
valore iniziale di un attributo, 186
valore posizionale, 638, 639
valore segnale, 61
valore sentinella, 61
valore simbolo, 638
valori di stato degli errori in uno stream, 624
variabile, 21, 344
variabile a sola lettura, 211
variabile costante, 211
variabile di controllo, 75

variabile globale, 157
variabile locale, 136, 157
variabile locale static, 158
variabile, indirizzo di memoria di una, 267
varianti di formattazione, 66
VAX VMS, 88
versione unaria degli operatori più (+) e meno (-), 66
vi, 12
visibilità, 78, 155, 157
visibilità a livello di blocco, 158
visibilità a livello di classe, 356
visibilità a livello di file, 158
visibilità a livello di funzione, 158
visibilità a livello di prototipo di funzione, 158
void, 170
vtable, 589

W

while, 50
Wirth, Niklaus, 10
workstation, 6
www.javasoft.com, 10

Z

zero in coda e punti decimali, 616

Finito di stampare nel mese di Marzo 2001
da Legoprint – Lavis, Trento

15,00